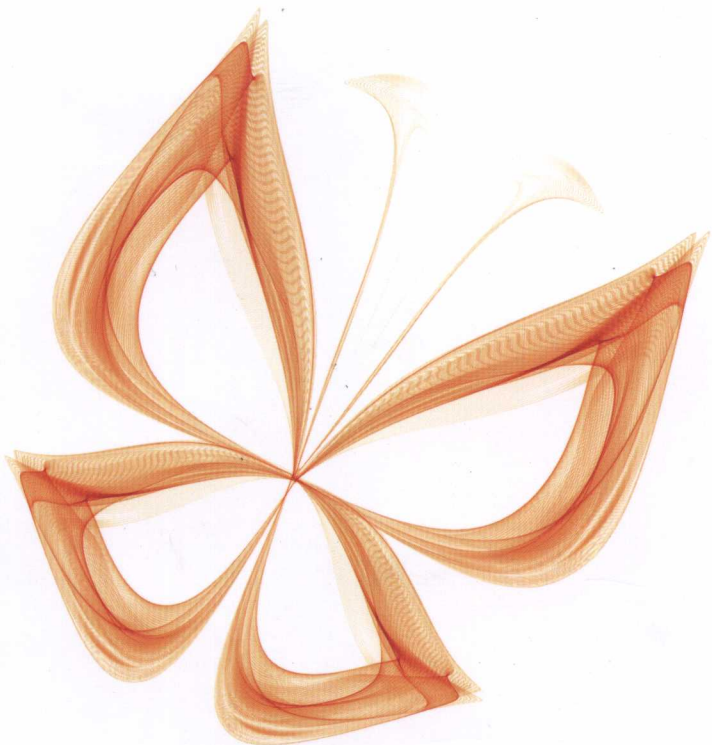
版权相关注意事项：
1、书籍版权归著者和出版社所有
2、本PDF来自于各个广泛的信息平台，经过整理而成
3、本PDF仅限用于非商业用途或者个人交流研究学习使用
4、本PDF获得者不得在互联网上以任何目的进行传播
5、如果觉得书籍内容很赞，请一定购买正版实体书，多多支持编写高质量的图书的作者和相应的出版社！当然，如果图书内容不堪入目，质量低下，你也可以选择狠狠滴撕裂本PDF
6、技术类书籍是拿来获取知识的，不是拿来收藏的，你得到了书籍不意味着你得到了知识，所以请不要得到书籍后就觉得沾沾自喜，要经常翻阅！！经常翻阅
7、请于下载PDF后24小时内研究使用并删掉本PDF

HZ BOOKS 华章IT

多位专家联袂推荐，360大数据专家撰写，基于Spark 2.1.0剖析架构与实现精髓

细化到方法级，提炼出多个流程图，立体呈现架构、环境、调度、存储、计算、部署、API七大核心设计

大数据技术丛书



The Art of Spark Kernel Design

Spark内核设计的艺术

架构设计与实现

耿嘉安◎著

机械工业出版社
China Machine Press

内容简介

多位专家联袂推荐，360大数据专家撰写，基于Spark 2.1.0剖析架构与实现精髓。细化到方法级，提炼出多个流程图，立体呈现架构、环境、调度、存储、计算、部署、API七大核心设计。本书一共有10章内容，主要包括以下部分。

准备部分（第1～2章）：简单介绍了Spark的环境搭建和基本原理。本部分通过详尽的描述，有效降低了读者进入Spark世界的门槛，同时能对Spark背景知识及整体设计有宏观的认识。

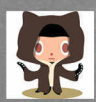
基础部分（第3～5章）：介绍Spark的基础设施（包括配置、RPC、度量等）、SparkContext的初始化、Spark执行所需要的环境等内容。经过此部分的学习，将能够对RPC框架的设计、执行环境的功能有深入的理解，这也是对核心内容了解的前提。

核心部分（第6～9章）：为Spark最核心的部分，包括存储体系、调度系统、计算引擎、部署模式等。通过本部分的学习，读者将充分了解Spark的数据处理体系细节，能够对Spark核心功能进行扩展、性能优化以及对线上问题进行精准排查。

API部分（第10章）：这部分主要对Spark的新老API进行对比，对新API进行简单介绍。

仅供非商业用途或交流学习使用

华章IT
HZBOOKS | Information Technology



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

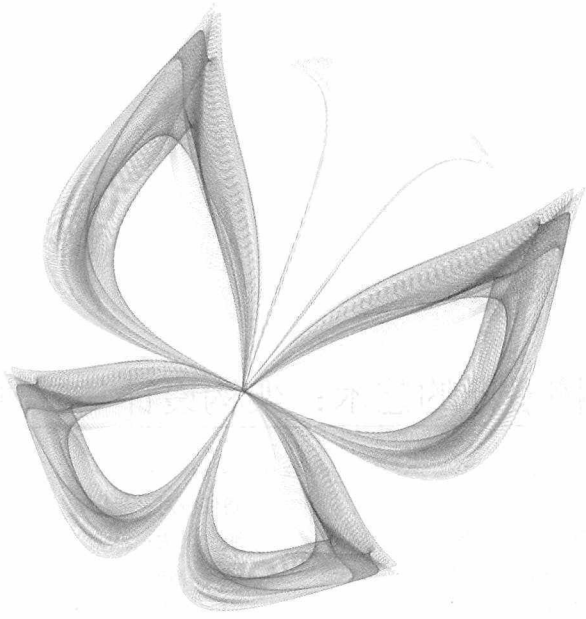
大数据技术丛书

The Art of Spark Kernel Design

Spark内核设计的艺术

架构设计与实现

耿嘉安◎著



机械工业出版社
China Machine Press

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

图书在版编目（CIP）数据

Spark 内核设计的艺术：架构设计与实现 / 耿嘉安著 . —北京：机械工业出版社，2017.12
（大数据技术丛书）

ISBN 978-7-111-58439-1

I. S… II. 耿… III. 数据处理软件 IV. TP274

中国版本图书馆 CIP 数据核字（2017）第 278308 号



Spark 内核设计的艺术：架构设计与实现

出版发行：机械工业出版社（北京市西城区百万庄大街 22 号　邮政编码：100037）
责任编辑：高婧雅　　责任校对：殷　虹
印　　刷：中国电影出版社印刷厂　　版　　次：2018 年 1 月第 1 版第 1 次印刷
开　　本：186mm × 240mm　1/16　　印　　张：44
书　　号：ISBN 978-7-111-58439-1　　定　　价：139.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换
客服热线：（010）88379426　88361066　　投稿热线：（010）88379604
购书热线：（010）68326294　88379649　68995259　　读者信箱：hzit@hzbook.com

版权所有 · 侵权必究
封底无防伪标均为盗版
本书法律顾问：北京大成律师事务所　韩光 / 邹晓东

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

Praise 本书赞誉

当年我在英国从事大数据工作，会经常去硅谷拜访大数据公司。其中最重要一个公司就是 Spark 创始人创建的 Databricks 了，最早一次是 2013 年 10 月，彼时 Databricks 刚起步，新办公室也尚在筹备。

4 年过去了，我们在大数据、流计算、图计算、分布式机器学习、深度学习等领域有了越来越多的高质量开源选择，但是 Spark 仍然是数据科学家们用得最多的工具之一，了解一点 Spark 底层技术的人都不得不对 Spark 的设计及其分布式计算的理论基础表示由衷敬佩。

本书对 Spark 内部高度抽象的数据结构 RDD、分布式 DAG 调度器 / 驱动器，以及高效的基于 Non-blocking IO 分布式计算框架 Akka/Netty 等内核设计进行了深度剖析，不可多得，是适合大型分布式计算架构师和资深开源贡献者阅读的参考书。

——蔡栋，万达网络科技集团总裁助理兼首席数据官、首席架构师



大数据技术生态其实是一个千姿百态的江湖。从学习技术的角度，最重要的是能将厚变薄，将纷繁复杂的信息进行归类和抽象。对应到大数据技术体系，虽然各种技术百花齐放，层出不穷，但大数据技术本质上无非解决 4 个核心问题：存储，计算，查询，挖掘。而 Spark 发展的短短几年，以迅雷不及掩耳之势推出 RDD、Spark Streaming、Spark SQL、GraphX、MLlib 等一系列模块，震撼了大数据圈。这本书结合了最新 Spark 2.x 版本，在设计思路和代码解析上做了很好的平衡，让开源代码爱好者，喜欢研究源码的同学汲取到一些阅读源码的方法。

——董飞，datatist 首席运营官、前 linkedin 资深工程师

初读本书有种似曾相识的感觉，Spark 还是那个 Spark，但是本书多了一些岁月的痕迹，在技术之上多了一些艺术，也更加注重读者的口味。大数据的书很多，能够写出艺术味道的不多，本书应该可以让你在大数据漫漫征途之中对价值多了一重思考，也可以让你在大数据之巅的惊天骇浪中多了一座灯塔。

——于俊，科大讯飞大数据专家

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

制度信息化，信息工具化，Spark为大数据产业落地提供有力的技术支撑工具！它以内存计算为核心，以其通用、快速和完整的数据工具形成了一个强有竞争力的数据生态圈，成为大数据技术解决方案非常优秀的一个部分，越来越多企业应用部署Spark。本书为那些想要成为一名合格的Spark工程师，或者致力于成为大数据行业的技术管理人才提供了很好的学习途径。相信读者只要掌握一门Spark技术，就能在大数据的海洋中遨游。感谢笔者为大数据产业做出的贡献！

——张涵诚，中关村大数据交易产业联盟副秘书长

本书对Spark原理的讲解与剖析都极具学习意义，作者细致分析了Spark源码的每一个关键细节，对初级用户及中高级用户都有指导意义。

——王欢，上海添锡信息技术有限公司技术总监



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

Preface 前 言

为什么写这本书

给本书写前言时，让我想起了两年前给《深入理解 Spark：核心思想与源码分析》[⊖]一书写前言的经历。我不禁想起崔护的《题都城南庄》这首诗，诗的内容是：

去年今日此门中，人面桃花相映红。
人面不知何处去，桃花依旧笑春风。

从核心思想和架构来看，Spark 依然是那个 Spark，但是我已经找了一个新的“东家”。我的年龄不知不觉中又长了两岁，Spark 也在大数据领域从“新贵”变成了“老人”。Spark 的版本从 0.x.x 到 2.x.x 基本上也是用了两年时间。



自从《深入理解 Spark：核心思想与源码分析》一书出版后，引起了一些市场反响，更难得的是得到了很多读者的反馈。一些热心的读者通过微信或者邮件向我指出了书中内容的很多不足之处，包括错别字、错误的描述、代码分析有点像流水账、提纲挈领的内容偏少、代码版本过低等。一些错误在修订的版本中得到了解决，有些修正的内容则通过单独写博客来补充。在与读者的沟通过程中，也纠正了我对一些问题的理解偏差。再次深深地感谢广大读者的支持与帮助！

一些读者对《深入理解 Spark：核心思想与源码分析》一书的内容非常肯定，希望能够出第 2 版，高婧雅编辑也一再“怂恿”我，但是我一直没有写第 2 版的打算。我当时希望有人能够以更好的方式写一本介绍和分析 Spark 2.0 版本的源码分析书籍，因为我感觉之前的写作方式的确不是很好。在我心中一直有个矛盾：如果源码太少，源码分析的书籍将退化成单纯讲原理的书籍，对于想深入理解 Spark 实现的读者来说这是不够的；如果源码太多，又让人有堆砌代码或者“混”篇幅的感觉。很多源码分析的书只是简单说说接口或者方法的功能，让人始终有种“雾里看花”的感觉。所以我一直很期待能有更好的方式来写作源码分析类的书。

在一年多的等待中，我始终没有发现类似书籍的出现，于是我打算再做一次尝试。这

⊖ 该书由机械工业出版社出版，书号 978-7-111-52234-8。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

次摈弃了《深入理解 Spark：核心思想与源码分析》一书中按照代码执行流程分析的方式，改为先从整体上介绍一个系统，然后逐个分析每个组件的功能，最后将这些组件之间的关系用流程图的方式串联起来。本书的写作方式依然犯有代码过多的“毛病”，但我还是期待本书能带来一些新的气象。

本书的主要特色

- 按照源码分析的习惯设计，从脚本分析到初始化，再到核心内容。整个过程遵循由浅入深的基本思路。
- 每一章先对本章的内容有个总体介绍，然后深入分析各个组件的实现原理，最后将各个组件之间的关系通过执行流程来展现。
- 本书尽可能地用图来展示原理，以加速读者对内容的掌握。
- 本书讲解的很多实现及原理都值得借鉴，可以帮助读者提升架构设计、程序设计等方面的能力。
- 本书尽可能保留较多的源码，以便于初学者能够在脱离办公环境的地方（如地铁、公交等），也能轻松阅读。

读者对象



源码阅读是一项苦差事，人力和时间成本都很高，尤其对于刚刚接触 Spark 的人来说更是如此。本书尽可能保留源码，使得分析过程不至于产生跳跃感，目的是降低大多数人的学习门槛。如果你是从事 IT 工作 1 ～ 3 年的新人或者希望开始学习 Spark 的核心知识，本书非常适合你。如果你已经对 Spark 有所了解或者已经使用它，还想进一步提高自己，那么本书更适合你。如果你是一个开发新手，对 Java、Linux 等基础知识还不是很了解的话，本书可能不太适合你。如果你已经对 Spark 有深入的研究，本书也许可以作为你的参考资料。

总体来说，本书适合以下人群：

- 已经了解过 Spark，但还想深入理解 Spark 实现原理的人；
- 大数据技术爱好者；
- 对性能优化和部署方案感兴趣的运维工程师与架构师；
- 开源代码爱好者，喜欢研究源码的同学可以通过本书学到一些阅读源码的方式、方法。

本书不会教你如何开发 Spark 应用程序，而只拿 word count 的经典例子做演示。本书会简单介绍 Hadoop MapReduce、Hadoop YARN、Mesos、Alluxio（Tachyon）、ZooKeeper、HDFS、Akka、Jetty、Netty，但不会过多介绍这些框架的使用，因为市场上已经有丰富的书籍供读者挑选。本书也不会过多介绍 Scala、Java、Shell 的语法，读者可以在市场上选择适合自己的书籍阅读。本书将无比适合那些想要破解“潘多拉魔盒”的人！

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

如何阅读本书

本书一共有 10 章内容，主要包括以下部分。

准备部分（第 1～2 章）：简单介绍了 Spark 的环境搭建和基本原理，帮助读者了解一些背景知识。

基础部分（第 3～5 章）：介绍 Spark 的基础设施、SparkContext 的初始化、Spark 执行环境等内容。

核心部分（第 6～9 章）：这是 Spark 最为核心的部分，包括存储体系、调度系统、计算引擎、部署模式等。

API 部分（第 10 章）：这部分主要对 Spark 的新旧 API 进行对比，对新 API 进行介绍。

本书最后的附录中还包括一些内容：附录 A 介绍的是 Spark 中最常用的工具类 Utils；附录 B 是 Akka 的简介；附录 C 为 Jetty 的简介和工具类 JettyUtils 的介绍；附录 D 为 Metrics 库的简介和 Metrics 中部分 API 的介绍；附录 E 演示了 Hadoop 1.0 版本中的 word count 例子；附录 F 介绍了工具类 CommandUtils 的常用方法；附录 G 是关于 Netty 的简介和工具类 NettyUtils 的介绍；附录 H 是对 Spark 中的 RPC 工具类 RpcUtils 的介绍。

为了降低读者阅读理解 Spark 源码的门槛，本书尽可能保留源码实现。本书以 Spark 2.1.0 版本为主，有兴趣的读者也可按照本书的方式，阅读 Spark 的最新源码。



勘误

本书内容很多，限于笔者水平有限，书中内容难免有错误之处。如果你对本书有任何问题或者意见，都可以通过邮箱 beliefer@163.com 或者博客 http://blog.csdn.net/beliefer 联系我，给我提交你的建议或者想法，我将怀着一颗谦卑之心与大家共同进步。

致谢

感谢我们生活在信息时代，让我们有机会接触互联网与大数据；感谢父母多年来在学习、工作及生活上的帮助与支持；感谢妻子在生活中的照顾和谦让。

感谢高婧雅编辑给予本书出版的大力支持与帮助。

感谢我在大数据路上的领路人——和仲；感谢热衷于技术的王欢对本书内容提出的宝贵建议；感谢对本书内容进行审阅的余尧尧和马晓波；感谢对本书内容有过帮助的读者朋友们。

耿嘉安

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

目　录 Contents

本书赞誉
前言

第 1 章　环境准备 ····· 1

1.1　运行环境准备 ····· 2
1.1.1　安装 JDK ····· 2
1.1.2　安装 Scala ····· 2
1.1.3　安装 Spark ····· 3
1.2　Spark 初体验 ····· 4
1.2.1　运行 spark-shell ····· 4
1.2.2　执行 word count ····· 5
1.2.3　剖析 spark-shell ····· 9
1.3　阅读环境准备 ····· 14
1.3.1　安装 SBT ····· 15
1.3.2　安装 Git ····· 15
1.3.3　安装 Eclipse Scala IDE 插件 ····· 15
1.4　Spark 源码编译与调试 ····· 17
1.5　小结 ····· 23

第 2 章　设计理念与基本架构 ····· 24

2.1　初识 Spark ····· 25
2.1.1　Hadoop MRv1 的局限 ····· 25
2.1.2　Spark 的特点 ····· 26
2.1.3　Spark 使用场景 ····· 28
2.2　Spark 基础知识 ····· 29
2.3　Spark 基本设计思想 ····· 31
2.3.1　Spark 模块设计 ····· 32
2.3.2　Spark 模型设计 ····· 34
2.4　Spark 基本架构 ····· 36
2.5　小结 ····· 38

第 3 章　Spark 基础设施 ····· 39

3.1　Spark 配置 ····· 40
3.1.1　系统属性中的配置 ····· 40
3.1.2　使用 SparkConf 配置的 API ····· 41
3.1.3　克隆 SparkConf 配置 ····· 42
3.2　Spark 内置 RPC 框架 ····· 42
3.2.1　RPC 配置 TransportConf ····· 45
3.2.2　RPC 客户端工厂 TransportClientFactory ····· 47
3.2.3　RPC 服务端 TransportServer ····· 53
3.2.4　管道初始化 ····· 56
3.2.5　TransportChannelHandler 详解 ····· 57
3.2.6　服务端 RpcHandler 详解 ····· 63
3.2.7　服务端引导程序 TransportServerBootstrap ····· 68



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

3.2.8 客户端 TransportClient 详解 …… 71
3.3 事件总线 …… 78
3.3.1 ListenerBus 的继承体系 …… 79
3.3.2 SparkListenerBus 详解 …… 80
3.3.3 LiveListenerBus 详解 …… 83
3.4 度量系统 …… 87
3.4.1 Source 继承体系 …… 87
3.4.2 Sink 继承体系 …… 89
3.5 小结 …… 92

第 4 章 SparkContext 的初始化 …… 93

4.1 SparkContext 概述 …… 94
4.2 创建 Spark 环境 …… 97
4.3 SparkUI 的实现 …… 100
4.3.1 SparkUI 概述 …… 100
4.3.2 WebUI 框架体系 …… 102
4.3.3 创建 SparkUI …… 107
4.4 创建心跳接收器 …… 111
4.5 创建和启动调度系统 …… 112
4.6 初始化块管理器 BlockManager …… 114
4.7 启动度量系统 …… 114
4.8 创建事件日志监听器 …… 115
4.9 创建和启动 ExecutorAllocation-Manager …… 116
4.10 ContextCleaner 的创建与启动 …… 120
4.10.1 创建 ContextCleaner …… 120
4.10.2 启动 ContextCleaner …… 120
4.11 额外的 SparkListener 与启动事件总线 …… 122
4.12 Spark 环境更新 …… 123
4.13 SparkContext 初始化的收尾 …… 127
4.14 SparkContext 提供的常用方法 …… 128
4.15 SparkContext 的伴生对象 …… 130
4.16 小结 …… 131

第 5 章 Spark 执行环境 …… 132

5.1 SparkEnv 概述 …… 133
5.2 安全管理器 SecurityManager …… 133
5.3 RPC 环境 …… 135
5.3.1 RPC 端点 RpcEndpoint …… 136
5.3.2 RPC 端点引用 RpcEndpointRef … 139
5.3.3 创建传输上下文 TransportConf … 142
5.3.4 消息调度器 Dispatcher …… 142
5.3.5 创建传输上下文 Transport-Context …… 154
5.3.6 创建传输客户端工厂 Transport-ClientFactory …… 159
5.3.7 创建 TransportServer …… 160
5.3.8 客户端请求发送 …… 162
5.3.9 NettyRpcEnv 中的常用方法 …… 173
5.4 序列化管理器 SerializerManager …… 175
5.5 广播管理器 BroadcastManager …… 178
5.6 map 任务输出跟踪器 …… 185
5.6.1 MapOutputTracker 的实现 …… 187
5.6.2 MapOutputTrackerMaster 的实现原理 …… 191
5.7 构建存储体系 …… 199
5.8 创建度量系统 …… 201
5.8.1 MetricsConfig 详解 …… 203
5.8.2 MetricsSystem 中的常用方法 … 207
5.8.3 启动 MetricsSystem …… 209
5.9 输出提交协调器 …… 211
5.9.1 OutputCommitCoordinator-Endpoint 的实现 …… 211



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

5.9.2 OutputCommitCoordinator 的实现 ····· 212
5.9.3 OutputCommitCoordinator 的工作原理 ····· 216
5.10 创建 SparkEnv ····· 217
5.11 小结 ····· 217

第 6 章 存储体系 ····· 219

6.1 存储体系概述 ····· 220
6.1.1 存储体系架构 ····· 220
6.1.2 基本概念 ····· 222
6.2 Block 信息管理器 ····· 227
6.2.1 Block 锁的基本概念 ····· 227
6.2.2 Block 锁的实现 ····· 229
6.3 磁盘 Block 管理器 ····· 234
6.3.1 本地目录结构 ····· 234
6.3.2 DiskBlockManager 提供的方法 ····· 236
6.4 磁盘存储 DiskStore ····· 239
6.5 内存管理器 ····· 242
6.5.1 内存池模型 ····· 243
6.5.2 StorageMemoryPool 详解 ····· 244
6.5.3 MemoryManager 模型 ····· 247
6.5.4 UnifiedMemoryManager 详解 ····· 250
6.6 内存存储 MemoryStore ····· 252
6.6.1 MemoryStore 的内存模型 ····· 253
6.6.2 MemoryStore 提供的方法 ····· 255
6.7 块管理器 BlockManager ····· 265
6.7.1 BlockManager 的初始化 ····· 265
6.7.2 BlockManager 提供的方法 ····· 266
6.8 BlockManagerMaster 对 BlockManager 的管理 ····· 285
6.8.1 BlockManagerMaster 的职责 ····· 285
6.8.2 BlockManagerMasterEndpoint 详解 ····· 286
6.8.3 BlockManagerSlaveEndpoint 详解 ····· 289
6.9 Block 传输服务 ····· 290
6.9.1 初始化 NettyBlockTransferService ····· 291
6.9.2 NettyBlockRpcServer 详解 ····· 292
6.9.3 Shuffle 客户端 ····· 296
6.10 DiskBlockObjectWriter 详解 ····· 305
6.11 小结 ····· 308

第 7 章 调度系统 ····· 309

7.1 调度系统概述 ····· 310
7.2 RDD 详解 ····· 312
7.2.1 为什么需要 RDD ····· 312
7.2.2 RDD 实现的初次分析 ····· 313
7.2.3 RDD 依赖 ····· 316
7.2.4 分区计算器 Partitioner ····· 318
7.2.5 RDDInfo ····· 320
7.3 Stage 详解 ····· 321
7.3.1 ResultStage 的实现 ····· 322
7.3.2 ShuffleMapStage 的实现 ····· 323
7.3.3 StageInfo ····· 324
7.4 面向 DAG 的调度器 DAGScheduler ····· 326
7.4.1 JobListener 与 JobWaiter ····· 326
7.4.2 ActiveJob 详解 ····· 328
7.4.3 DAGSchedulerEventProcessLoop 的简要介绍 ····· 328
7.4.4 DAGScheduler 的组成 ····· 329
7.4.5 DAGScheduler 提供的常用方法 ····· 330

仅供非商业用途或交流学习使用

7.4.6 DAGScheduler 与 Job 的提交 …… 334
7.4.7 构建 Stage …… 337
7.4.8 提交 ResultStage …… 341
7.4.9 提交还未计算的 Task …… 343
7.4.10 DAGScheduler 的调度流程 …… 347
7.4.11 Task 执行结果的处理 …… 348
7.5 调度池 Pool …… 351
7.5.1 调度算法 …… 352
7.5.2 Pool 的实现 …… 354
7.5.3 调度池构建器 …… 357
7.6 任务集合管理器 TaskSetManager …… 363
7.6.1 Task 集合 …… 363
7.6.2 TaskSetManager 的成员属性 …… 364
7.6.3 调度池与推断执行 …… 366
7.6.4 Task 本地性 …… 370
7.6.5 TaskSetManager 的常用方法 …… 373
7.7 运行器后端接口 LauncherBackend …… 383
7.7.1 BackendConnection 的实现 …… 384
7.7.2 LauncherBackend 的实现 …… 386
7.8 调度后端接口 SchedulerBackend …… 389
7.8.1 SchedulerBackend 的定义 …… 389
7.8.2 LocalSchedulerBackend 的实现分析 …… 390
7.9 任务结果获取器 TaskResultGetter …… 394
7.9.1 处理成功的 Task …… 394
7.9.2 处理失败的 Task …… 396
7.10 任务调度器 TaskScheduler …… 397
7.10.1 TaskSchedulerImpl 的属性 …… 397
7.10.2 TaskSchedulerImpl 的初始化 …… 399
7.10.3 TaskSchedulerImpl 的启动 …… 399
7.10.4 TaskSchedulerImpl 与 Task 的提交 …… 400
7.10.5 TaskSchedulerImpl 与资源分配 …… 402
7.10.6 TaskSchedulerImpl 的调度流程 …… 405
7.10.7 TaskSchedulerImpl 对执行结果的处理 …… 406
7.10.8 TaskSchedulerImpl 的常用方法 …… 409
7.11 小结 …… 412

第 8 章 计算引擎 …… 413

8.1 计算引擎概述 …… 414
8.2 内存管理器与执行内存 …… 417
8.2.1 ExecutionMemoryPool 详解 …… 417
8.2.2 MemoryManager 模型与执行内存 …… 420
8.2.3 UnifiedMemoryManager 与执行内存 …… 421
8.3 内存管理器与 Tungsten …… 423
8.3.1 MemoryBlock 详解 …… 423
8.3.2 MemoryManager 模型与 Tungsten …… 425
8.3.3 Tungsten 的内存分配器 …… 425
8.4 任务内存管理器 …… 431
8.4.1 TaskMemoryManager 详解 …… 431
8.4.2 内存消费者 …… 439
8.4.3 执行内存整体架构 …… 441
8.5 Task 详解 …… 443
8.5.1 任务上下文 TaskContext …… 443
8.5.2 Task 的定义 …… 446
8.5.3 ShuffleMapTask 的实现 …… 449
8.5.4 ResultTask 的实现 …… 450
8.6 IndexShuffleBlockResolver 详解 …… 451
8.7 采样与估算 …… 455
8.7.1 SizeTracker 的实现分析 …… 455



仅供非商业用途或交流学习使用

8.7.2 SizeTracker 的工作原理 ····· 457
8.8 特质 WritablePartitionedPair-Collection ····· 458
8.9 AppendOnlyMap 的实现分析 ····· 460
8.9.1 AppendOnlyMap 的容量增长 ····· 461
8.9.2 AppendOnlyMap 的数据更新 ····· 462
8.9.3 AppendOnlyMap 的缓存聚合算法 ····· 464
8.9.4 AppendOnlyMap 的内置排序 ····· 466
8.9.5 AppendOnlyMap 的扩展 ····· 467
8.10 PartitionedPairBuffer 的实现分析 ····· 469
8.10.1 PartitionedPairBuffer 的容量增长 ····· 469
8.10.2 PartitionedPairBuffer 的插入 ····· 470
8.10.3 PartitionedPairBuffer 的迭代器 ····· 471
8.11 外部排序器 ····· 472
8.11.1 ExternalSorter 详解 ····· 473
8.11.2 ShuffleExternalSorter 详解 ····· 487
8.12 Shuffle 管理器 ····· 490
8.12.1 ShuffleWriter 详解 ····· 491
8.12.2 ShuffleBlockFetcherIterator 详解 ····· 502
8.12.3 BlockStoreShuffleReader 详解 ····· 510
8.12.4 SortShuffleManager 详解 ····· 513
8.13 map 端与 reduce 端的 Shuffle 组合 ····· 516
8.14 小结 ····· 519

第 9 章 部署模式 ····· 520

9.1 心跳接收器 HeartbeatReceiver ····· 521
9.2 Executor 的实现分析 ····· 527
9.2.1 Executor 的心跳报告 ····· 528
9.2.2 运行 Task ····· 530
9.3 local 部署模式 ····· 535
9.4 持久化引擎 PersistenceEngine ····· 537
9.4.1 基于文件系统的持久化引擎 ····· 539
9.4.2 基于 ZooKeeper 的持久化引擎 ····· 541
9.5 领导选举代理 ····· 542
9.6 Master 详解 ····· 546
9.6.1 启动 Master ····· 549
9.6.2 检查 Worker 超时 ····· 553
9.6.3 被选举为领导时的处理 ····· 554
9.6.4 一级资源调度 ····· 558
9.6.5 注册 Worker ····· 568
9.6.6 更新 Worker 的最新状态 ····· 570
9.6.7 处理 Worker 的心跳 ····· 570
9.6.8 注册 Application ····· 571
9.6.9 处理 Executor 的申请 ····· 573
9.6.10 处理 Executor 的状态变化 ····· 573
9.6.11 Master 的常用方法 ····· 574
9.7 Worker 详解 ····· 578
9.7.1 启动 Worker ····· 581
9.7.2 向 Master 注册 Worker ····· 584
9.7.3 向 Master 发送心跳 ····· 589
9.7.4 Worker 与领导选举 ····· 591
9.7.5 运行 Driver ····· 593
9.7.6 运行 Executor ····· 594
9.7.7 处理 Executor 的状态变化 ····· 599
9.8 StandaloneAppClient 实现 ····· 600
9.8.1 ClientEndpoint 的实现分析 ····· 601
9.8.2 StandaloneAppClient 的实现分析 ····· 606
9.9 StandaloneSchedulerBackend 的实现分析 ····· 607

仅供非商业用途或交流学习使用

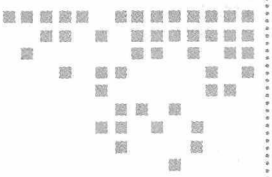
9.9.1 StandaloneSchedulerBackend 的属性 ····· 607
9.9.2 DriverEndpoint 的实现分析 ····· 609
9.9.3 StandaloneSchedulerBackend 的启动 ····· 614
9.9.4 StandaloneSchedulerBackend 的停止 ····· 617
9.9.5 StandaloneSchedulerBackend 与资源分配 ····· 618
9.10 CoarseGrainedExecutorBackend 详解 ····· 619
9.10.1 CoarseGrainedExecutorBackend 进程 ····· 620
9.10.2 CoarseGrainedExecutorBackend 的功能分析 ····· 622
9.11 local-cluster 部署模式 ····· 625
9.11.1 启动本地集群 ····· 625
9.11.2 local-cluster 部署模式的启动过程 ····· 627
9.11.3 local-cluster 部署模式下 Executor 的分配过程 ····· 628
9.11.4 local-cluster 部署模式下的任务提交执行过程 ····· 629
9.12 Standalone 部署模式 ····· 631
9.12.1 Standalone 部署模式的启动过程 ····· 632
9.12.2 Standalone 部署模式下 Executor 的分配过程 ····· 634
9.12.3 Standalone 部署模式的资源回收 ····· 635
9.12.4 Standalone 部署模式的容错机制 ····· 636
9.13 其他部署方案 ····· 639
9.13.1 YARN ····· 639
9.13.2 Mesos ····· 644
9.14 小结 ····· 646

第 10 章 Spark API ····· 647

10.1 基本概念 ····· 648
10.2 数据源 DataSource ····· 650
10.2.1 DataSourceRegister 详解 ····· 650
10.2.2 DataSource 详解 ····· 651
10.3 检查点的实现 ····· 655
10.3.1 CheckpointRDD 的实现 ····· 655
10.3.2 RDDCheckpointData 的实现 ····· 660
10.3.3 ReliableRDDCheckpointData 的实现 ····· 662
10.4 RDD 的再次分析 ····· 663
10.4.1 转换 API ····· 663
10.4.2 动作 API ····· 665
10.4.3 检查点 API 的实现分析 ····· 667
10.4.4 迭代计算 ····· 669
10.5 数据集合 Dataset ····· 671
10.6 DataFrameReader 详解 ····· 673
10.7 SparkSession 详解 ····· 676
10.7.1 SparkSession 的构建器 Builder ····· 676
10.7.2 SparkSession 的 API ····· 679
10.8 word count 例子 ····· 679
10.8.1 Job 准备阶段 ····· 680
10.8.2 Job 的提交与调度 ····· 685
10.9 小结 ····· 689

附录 ····· 690

仅供非商业用途或交流学习使用



第 1 章 Chapter 1

环境准备

“凡事豫则立，不豫则废。言前定，则不跲；事前定，则不困。”

——《礼记·中庸》



本章导读

学习一个工具的最好途径，就是使用它。这就好比《极品飞车》玩得好的同学，未必真的会开车，要学习车的驾驶技能，就必须用手触摸方向盘，用脚感受刹车与油门的力道。在 IT 领域，在深入了解一个系统的原理和实现细节之前，应当先准备好它的运行环境或者源码阅读环境。如果能在实际环境下安装和运行 Spark，显然能够提升读者对 Spark 的一些感受，对系统能有个大体的印象，有经验的工程师甚至能够猜出一些 Spark 在实现过程中采用的设计模式和编程模型。

当你通过一些途径知道了系统的原理之后，难道不会问问自己：这是怎么做到的？如果只是游走于系统使用、原理了解的层面，是永远不可能真正理解整个系统的。很多 IDE 本身带有调试的功能，每当你阅读源码，陷入重围时，调试能让我们更加理解运行期的系统。如果没有调试功能，不敢想象阅读源码的难度有多大。

本章旨在帮助读者搭建基本的运行环境、构建源码学习环境。主要包括以下内容。

- 在 Linux 中搭建基本的 Spark 执行环境。
- spark-shell 的基本使用。
- spark-shell 的源码剖析。
- 在 Mac OS 中搭建 Spark 源码阅读环境。

仅供非商业用途或交流学习使用

1.1 运行环境准备

考虑到大部分公司开发和生产环境都采用 Linux 操作系统，所以笔者选用了 64 位的 Linux。在正式安装 Spark 之前，先要找台好机器。为什么？因为笔者在安装、编译、调试的过程中发现 Spark 非常耗费内存，如果机器配置太低，恐怕会运行不起来。Spark 的开发语言是 Scala，而 Scala 需要运行在 JVM 之上，因而搭建 Spark 的运行环境应该包括 JDK 和 Scala。

1.1.1 安装 JDK

自 Spark 2.0.0 版本开始，Spark 已经放弃了对 Java 7 的支持，所以需要选择 Java 8。我们还需要使用命令 getconf LONG_BIT 查看 Linux 机器是 32 位还是 64 位，然后下载相应版本的 JDK 并安装。

下载地址：http://www.oracle.com/technetwork/java/javase/downloads/index.html。

配置环境：

```
cd ~
vim .bash_profile
```

添加如下配置：

```
export JAVA_HOME=/opt/java
export PATH=$PATH:$JAVA_HOME/bin
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
```

输入以下命令使环境变量快速生效：

```
source .bash_profile
```

安装完毕后，使用 java –version 命令查看，确认安装正常，如图 1-1 所示。

```
EBJ1266:~ user$ java -version
java version "1.8.0_102"
Java(TM) SE Runtime Environment (build 1.8.0_102-b14)
Java HotSpot(TM) 64-Bit Server VM (build 25.102-b14, mixed mode)
```

图 1-1 查看 java 安装是否正常

1.1.2 安装 Scala

由于从 Spark 2.0.0 开始，Spark 默认使用 Scala 2.11 来编译、打包，不再是以前的 Scala 2.10，所以我们需要下载 Scala 2.11。

下载地址：http://www.scala-lang.org/download/。

选择 Scala 2.11 的版本进行下载，下载方法如下：

```
wget https://downloads.lightbend.com/scala/2.11.8/scala-2.11.8.tgz
```

移动到选好的安装目录，例如：

仅供非商业用途或交流学习使用


```
mv scala-2.11.8.tgz ~/install/
```

进入安装目录，执行以下命令：

```
chmod 755 scala-2.11.8.tgz
tar -xzvf scala-2.11.8.tgz
```

配置环境：

```
cd ~
vim .bash_profile
```

添加如下配置：

```
export SCALA_HOME=$HOME/install/scala-2.11.8
export PATH=$SCALA_HOME/bin:$PATH
```

输入以下命令使环境变量快速生效：

```
source .bash_profile
```

安装完毕后键入 scala，进入 scala 命令行，以确认安装正常，如图 1-2 所示。



图 1-2 进入 Scala 命令行



1.1.3 安装 Spark

Spark 进入 2.0 时代之后，目前一共有两个大的版本：一个是 Spark 2.0.0，一个是 Spark 2.1.0。本书选择 Spark 2.1.0。

下载地址：http://spark.apache.org/downloads.html。

下载方法如下：

```
wget http://d3kbcqa49mib13.cloudfront.net/spark-2.1.0-bin-hadoop2.6.tgz
```

移动到选好的安装目录，例如：

```
mv spark-2.1.0-bin-hadoop2.6.tgz ~/install/
```

进入安装目录，执行以下命令：

```
chmod 755 spark-2.1.0-bin-hadoop2.6.tgz
tar -xzvf spark-2.1.0-bin-hadoop2.6.tgz
```

配置环境：

```
cd ~
vim .bash_profile
```

添加如下配置：

```
export SPARK_HOME=$HOME/install/spark-2.1.0-bin-hadoop2.6
export PATH=$SPARK_HOME/bin:$PATH
```

仅供非商业用途或交流学习使用

输入以下命令，使环境变量快速生效：

```
source .bash_profile
```

安装完毕后键入 spark-shell，进入 scala 命令行，以确认安装正常，如图 1-3 所示。

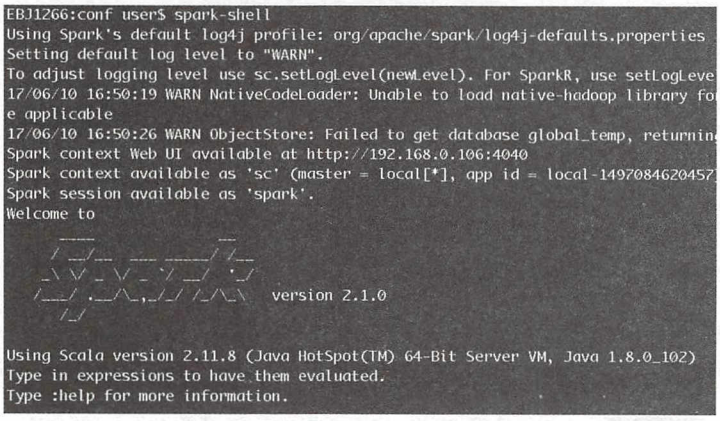


图 1-3　执行 spark-shell 进入 Scala 命令行

1.2　Spark 初体验



本节通过 Spark 的基本使用，让读者对 Spark 能有初步的认识，以引导读者逐步深入学习。

1.2.1　运行 spark-shell

图 1-3 中显示了很多信息，这里进行一些说明。

- 安装完 Spark 2.1.0 后，如果没有明确指定 log4j 的配置，那么 Spark 会使用 core 模块的 org/apache/spark/ 目录下的 log4j-defaults.properties 作为 log4j 的默认配置。log4j-defaults.properties 指定的 Spark 日志级别为 WARN。用户可以到 Spark 安装目录的 conf 文件夹下，从 log4j.properties.template 复制一份 log4j.properties 文件，并在其中增加自己想要的配置。
- 除了指定 log4j.properties 文件外，还可以在 spark-shell 命令行中通过 sc.setLogLevel(newLevel) 语句指定日志级别。
- SparkContext 的 Web UI 的地址是：http://192.168.0.106:4040。192.168.0.106 是作者本地安装 Spark 的机器的 IP 地址，4040 是 SparkContext 的 Web UI 的默认监听端口。
- 指定的部署模式（即 master）为 local[*]。当前应用（Application）的 ID 为 local-1497084620457。
- 可以在 spark-shell 命令行通过 sc 使用 SparkContext，通过 spark 使用 SparkSession。sc 和 spark 实际分别是 SparkContext 和 SparkSession 在 Spark REPL 中的变量名，

仅供非商业用途或交流学习使用

具体细节将在 1.2.3 节分析。

由于 Spark Core 的默认日志级别是 WARN，所以看到的信息不是很多。现在我们将 Spark 安装目录的 conf 文件夹下的 log4j.properties.template 通过如下命令复制出一份：

```
cp log4j.properties.template log4j.properties
```

并将 log4j.properties 中的 log4j.logger.org.apache.spark.repl.Main=WARN 修改为 log4j.logger.org.apache.spark.repl.Main=INFO，然后我们再次运行 spark-shell，将打印出更丰富的信息，如图 1-4 所示。

```
EBJ1266:conf user$ spark-shell
17/06/10 17:20:16 INFO SparkContext: Running Spark version 2.1.0
17/06/10 17:20:16 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using bui
e applicable
17/06/10 17:20:17 INFO SecurityManager: Changing view acls to: user
17/06/10 17:20:17 INFO SecurityManager: Changing modify acls to: user
17/06/10 17:20:17 INFO SecurityManager: Changing view acls groups to:
17/06/10 17:20:17 INFO SecurityManager: Changing modify acls groups to:
17/06/10 17:20:17 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled; users  w
Set(user); groups with view permissions: Set(); users  with modify permissions: Set(user); groups with modif
17/06/10 17:20:17 INFO Utils: Successfully started service 'sparkDriver' on port 50172.
17/06/10 17:20:17 INFO SparkEnv: Registering MapOutputTracker
17/06/10 17:20:17 INFO SparkEnv: Registering BlockManagerMaster
17/06/10 17:20:17 INFO BlockManagerMasterEndpoint: Using org.apache.spark.storage.DefaultTopologyMapper for
mation
17/06/10 17:20:17 INFO BlockManagerMasterEndpoint: BlockManagerMasterEndpoint up
17/06/10 17:20:17 INFO DiskBlockManager: Created local directory at /private/var/folders/fx/63g6ddhj2zn_n6kt
mgr-e6173e51-9073-448b-b9c6-9e8dc67aeeb4
17/06/10 17:20:18 INFO MemoryStore: MemoryStore started with capacity 366.3 MB
17/06/10 17:20:18 INFO SparkEnv: Registering OutputCommitCoordinator
17/06/10 17:20:18 INFO Utils: Successfully started service 'SparkUI' on port 4040.
17/06/10 17:20:18 INFO SparkUI: Bound SparkUI to 0.0.0.0, and started at http://192.168.0.106:4040
17/06/10 17:20:18 INFO Executor: Starting executor ID driver on host localhost
17/06/10 17:20:18 INFO Executor: Using REPL class URI: spark://192.168.0.106:50172/classes
17/06/10 17:20:18 INFO Utils: Successfully started service 'org.apache.spark.network.netty.NettyBlockTransfe
3.
17/06/10 17:20:18 INFO NettyBlockTransferService: Server created on 192.168.0.106:50173
17/06/10 17:20:18 INFO BlockManager: Using org.apache.spark.storage.RandomBlockReplicationPolicy for block r
17/06/10 17:20:18 INFO BlockManagerMaster: Registering BlockManager BlockManagerId(driver, 192.168.0.106, 50
17/06/10 17:20:18 INFO BlockManagerMasterEndpoint: Registering block manager 192.168.0.106:50173 with 366.3
(driver, 192.168.0.106, 50173, None)
17/06/10 17:20:18 INFO BlockManagerMaster: Registered BlockManager BlockManagerId(driver, 192.168.0.106, 501
17/06/10 17:20:18 INFO BlockManager: Initialized BlockManager: BlockManagerId(driver, 192.168.0.106, 50173,
17/06/10 17:20:19 INFO SharedState: Warehouse path is 'file:/Users/user/install/spark-2.1.0-bin-hadoop2.6/co
```

图 1-4 Spark 启动过程打印的部分信息

从图 1-4 展示的启动日志中我们可以看到 SecurityManager、SparkEnv、BlockManagerMasterEndpoint、DiskBlockManager、MemoryStore、SparkUI、Executor、NettyBlockTransferService、BlockManager、BlockManagerMaster 等信息。它们是做什么的？刚刚接触 Spark 的读者只需要知道这些信息即可，具体内容将在后边进行详细的介绍。

1.2.2 执行 word count

这一节，我们通过 word count 这个耳熟能详的例子来感受下 Spark 任务的执行过程。

仅供非商业用途或交流学习使用

启动 spark-shell 后，会打开 Scala 命令行，然后按照以下步骤输入脚本。

1）输入 val lines = sc.textFile("../README.md", 2)，以 Spark 安装目录下的 README.md 文件内容作为 word count 例子的数据源，执行结果如图 1-5 所示。

```
scala> val lines = sc.textFile("../README.md", 2)
17/06/10 17:31:32 INFO MemoryStore: Block broadcast_0 stored as values in memory (estimated size 208.6 KB, free 366.1 MB)
17/06/10 17:31:32 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in memory (estimated size 20.1 KB, free 366.1 MB)
17/06/10 17:31:32 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on 192.168.0.106:50173 (size: 20.1 KB, free: 366.3 MB)
17/06/10 17:31:32 INFO SparkContext: Created broadcast 0 from textFile at <console>:24
lines: org.apache.spark.rdd.RDD[String] = ../README.md MapPartitionsRDD[1] at textFile at <console>:24
```

图 1-5　步骤 1 执行结果

图 1-5 告诉我们，lines 的实际类型是 MapPartitionsRDD ⊖。

2）textFile 方法对文本文件是逐行读取的，我们需要输入 val words = lines.flatMap(line => line.split(" "))，将每行文本按照空格分隔以得到每个单词，执行结果如图 1-6 所示。

```
scala> val words = lines.flatMap(line => line.split(" "))
words: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2] at flatMap at <console>:26
```

图 1-6　步骤 2 执行结果

图 1-6 告诉我们，lines 在经过 flatMap 方法的转换后，得到的 words 的实际类型也是 MapPartitionsRDD。

3）对于得到的每个单词，通过输入 val ones = words.map(w => (w,1))，将每个单词的计数初始化为 1，执行结果如图 1-7 所示。

```
scala> val ones = words.map(w => (w,1))
ones: org.apache.spark.rdd.RDD[(String, Int)] = MapPartitionsRDD[3] at map at <console>:28
```

图 1-7　步骤 3 执行结果

图 1-7 告诉我们，words 在经过 map 方法的转换后，得到的 ones 的实际类型也是 MapPartitionsRDD。

4）输入 val counts = ones.reduceByKey(_ + _)，对单词进行计数值的聚合，执行结果如图 1-8 所示。

```
scala> val counts = ones.reduceByKey(_ + _)
17/06/10 17:38:01 INFO FileInputFormat: Total input paths to process : 1
counts: org.apache.spark.rdd.RDD[(String, Int)] = ShuffledRDD[4] at reduceByKey at <console>:30
```

图 1-8　步骤 4 执行结果

图 1-8 告诉我们，ones 在经过 reduceByKey 方法的转换后，得到的 counts 的实际类型是 ShuffledRDD。

⊖ 这里特别说明一点，在生成 MapPartitionsRDD 之前还实例化了 HadoopRDD，作为 MapPartitionsRDD 的上游 RDD。

仅供非商业用途或交流学习使用

5）输入 counts.foreach（println），将每个单词的计数值打印出来，作业的执行过程如图 1-9 和图 1-10 所示。作业的输出结果如图 1-11 所示。

```
scala> counts.foreach(println)
17/06/10 17:46:35 INFO SparkContext: Starting job: foreach at <console>:33
17/06/10 17:46:35 INFO DAGScheduler: Registering RDD 3 (map at <console>:28)
17/06/10 17:46:35 INFO DAGScheduler: Got job 0 (foreach at <console>:33) with 2 output partitions
17/06/10 17:46:35 INFO DAGScheduler: Final stage: ResultStage 1 (foreach at <console>:33)
17/06/10 17:46:35 INFO DAGScheduler: Parents of final stage: List(ShuffleMapStage 0)
17/06/10 17:46:35 INFO DAGScheduler: Missing parents: List(ShuffleMapStage 0)
17/06/10 17:46:35 INFO DAGScheduler: Submitting ShuffleMapStage 0 (MapPartitionsRDD[3] at map at <console>:28), which has no missi
ng parents
17/06/10 17:46:35 INFO MemoryStore: Block broadcast_1 stored as values in memory (estimated size 4.8 KB, free 366.1 MB)
17/06/10 17:46:35 INFO MemoryStore: Block broadcast_1_piece0 stored as bytes in memory (estimated size 2.7 KB, free 366.1 MB)
17/06/10 17:46:35 INFO BlockManagerInfo: Added broadcast_1_piece0 in memory on 192.168.0.106:50173 (size: 2.7 KB, free: 366.3 MB)
17/06/10 17:46:35 INFO SparkContext: Created broadcast 1 from broadcast at DAGScheduler.scala:996
17/06/10 17:46:35 INFO DAGScheduler: Submitting 2 missing tasks from ShuffleMapStage 0 (MapPartitionsRDD[3] at map at <console>:28
)
17/06/10 17:46:35 INFO TaskSchedulerImpl: Adding task set 0.0 with 2 tasks
17/06/10 17:46:35 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0, localhost, executor driver, partition 0, PROCESS_LOC
AL, 5997 bytes)
17/06/10 17:46:35 INFO TaskSetManager: Starting task 1.0 in stage 0.0 (TID 1, localhost, executor driver, partition 1, PROCESS_LOC
AL, 5997 bytes)
17/06/10 17:46:35 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
17/06/10 17:46:35 INFO Executor: Running task 1.0 in stage 0.0 (TID 1)
17/06/10 17:46:36 INFO HadoopRDD: Input split: file:/Users/user/install/spark-2.1.0-bin-hadoop2.6/README.md:1909+1909
17/06/10 17:46:36 INFO HadoopRDD: Input split: file:/Users/user/install/spark-2.1.0-bin-hadoop2.6/README.md:0+1909
17/06/10 17:46:36 INFO deprecation: mapred.tip.id is deprecated. Instead, use mapreduce.task.id
17/06/10 17:46:36 INFO deprecation: mapred.task.is.map is deprecated. Instead, use mapreduce.task.ismap
17/06/10 17:46:36 INFO deprecation: mapred.task.id is deprecated. Instead, use mapreduce.task.attempt.id
17/06/10 17:46:36 INFO deprecation: mapred.job.id is deprecated. Instead, use mapreduce.job.id
17/06/10 17:46:36 INFO deprecation: mapred.task.partition is deprecated. Instead, use mapreduce.task.partition
17/06/10 17:46:36 INFO Executor: Finished task 1.0 in stage 0.0 (TID 1). 1746 bytes result sent to driver
17/06/10 17:46:36 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 1746 bytes result sent to driver
17/06/10 17:46:36 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 371 ms on localhost (executor driver) (1/2)
```

图 1-9　步骤 5 执行过程第一部分



```
17/06/10 17:46:36 INFO TaskSetManager: Finished task 1.0 in stage 0.0 (TID 1) in 329 ms on localhost (executor driver) (2/2)
17/06/10 17:46:36 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool
17/06/10 17:46:36 INFO DAGScheduler: ShuffleMapStage 0 (map at <console>:28) finished in 0.407 s
17/06/10 17:46:36 INFO DAGScheduler: looking for newly runnable stages
17/06/10 17:46:36 INFO DAGScheduler: running: Set()
17/06/10 17:46:36 INFO DAGScheduler: waiting: Set(ResultStage 1)
17/06/10 17:46:36 INFO DAGScheduler: failed: Set()
17/06/10 17:46:36 INFO DAGScheduler: Submitting ResultStage 1 (ShuffledRDD[4] at reduceByKey at <console>:30), which has no missin
g parents
17/06/10 17:46:36 INFO MemoryStore: Block broadcast_2 stored as values in memory (estimated size 3.1 KB, free 366.1 MB)
17/06/10 17:46:36 INFO MemoryStore: Block broadcast_2_piece0 stored as bytes in memory (estimated size 1958.0 B, free 366.1 MB)
17/06/10 17:46:36 INFO BlockManagerInfo: Added broadcast_2_piece0 in memory on 192.168.0.106:50173 (size: 1958.0 B, free: 366.3 MB
)
17/06/10 17:46:36 INFO SparkContext: Created broadcast 2 from broadcast at DAGScheduler.scala:996
17/06/10 17:46:36 INFO DAGScheduler: Submitting 2 missing tasks from ResultStage 1 (ShuffledRDD[4] at reduceByKey at <console>:30)
17/06/10 17:46:36 INFO TaskSchedulerImpl: Adding task set 1.0 with 2 tasks
17/06/10 17:46:36 INFO TaskSetManager: Starting task 0.0 in stage 1.0 (TID 2, localhost, executor driver, partition 0, ANY, 5750 b
ytes)
17/06/10 17:46:36 INFO TaskSetManager: Starting task 1.0 in stage 1.0 (TID 3, localhost, executor driver, partition 1, ANY, 5750 b
ytes)
17/06/10 17:46:36 INFO Executor: Running task 0.0 in stage 1.0 (TID 2)
17/06/10 17:46:36 INFO Executor: Running task 1.0 in stage 1.0 (TID 3)
17/06/10 17:46:36 INFO ShuffleBlockFetcherIterator: Getting 2 non-empty blocks out of 2 blocks
17/06/10 17:46:36 INFO ShuffleBlockFetcherIterator: Getting 2 non-empty blocks out of 2 blocks
17/06/10 17:46:36 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 10 ms
17/06/10 17:46:36 INFO ShuffleBlockFetcherIterator: Started 0 remote fetches in 11 ms
```

图 1-10　步骤 5 执行过程第二部分

图 1-9 和图 1-10 展示了很多作业提交、执行的信息，这里挑选关键的内容进行介绍。

仅供非商业用途或交流学习使用


```
(Pi,1)
(that,2)
(protocols,1)
(a,8)
(or,3)
(high-level,1)
(name,1)
(Hadoop,,2)
(to,17)
(available,1)
((You,1)
(core,1)
(more,1)
(see,3)
(of,5)
(tools,1)
("local[N]",1)
(programs,2)
(option,1)
(package.),1)
(["Building,1)
(instance:,1)
(must,1)
(and,9)
(command,,2)
(system,1)
(Hadoop,3)
17/06/10 17:46:36 INFO Executor: Finished task 0.0 in stage 1.0 (TID 2). 1809 bytes result sent to driver
17/06/10 17:46:36 INFO Executor: Finished task 1.0 in stage 1.0 (TID 3). 1809 bytes result sent to driver
17/06/10 17:46:36 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 2) in 120 ms on localhost (executor driver) (1/2)
17/06/10 17:46:36 INFO TaskSetManager: Finished task 1.0 in stage 1.0 (TID 3) in 116 ms on localhost (executor driver) (2/2)
17/06/10 17:46:36 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all completed, from pool
17/06/10 17:46:36 INFO DAGScheduler: ResultStage 1 (foreach at <console>:33) finished in 0.122 s
17/06/10 17:46:36 INFO DAGScheduler: Job 0 finished: foreach at <console>:33, took 0.947761 s
```

图 1-11　步骤 5 输出结果



- SparkContext 为提交的 Job 生成的 ID 是 0。
- word count 例子一共产生了 4 个 RDD[⊖]，被划分为 ResultStage 和 ShuffleMapStage。ShuffleMapStage 的 ID 为 0，尝试号为 0。ResultStage 的 ID 为 1，尝试号也为 0。在 Spark 中，如果 Stage 没有执行完成，就会进行多次重试。Stage 无论是首次执行还是重试，都被视为是一次 Stage 尝试（stage attempt），每次尝试都有一个唯一的尝试号（attempt number）。
- 由于 Job 有两个分区，所以 ShuffleMapStage 和 ResultStage 都有两个 Task 被提交。每个 Task 也会有多次尝试，因而也有属于 Task 的尝试号。从图中看出，ShuffleMapStage 中的两个 Task 和 ResultStage 中的两个 Task 的尝试号也都是 0。
- HadoopRDD 则用于读取文件内容。

图 1-11 展示了单词计数的输出结果和最后打印的任务结束的日志信息。

笔者在本节介绍的 word count 例子是以 SparkContext 的 API 来实现的，读者也可以选择在 spark-shell 中通过运用 SparkSession 的 API 来实现。本书在第 10 章将介绍 Spark 源码自带的用 SparkSession 的 API 来实现的 word count 的 Java 应用程序。

⊖ 这里没有算入最上游的 HadoopRDD。

仅供非商业用途或交流学习使用

1.2.3 剖析 spark-shell

通过在 spark-shell 中执行 word count 的过程，让读者了解到可以使用 spark-shell 提交 Spark 作业。现在读者应该很想知道 spark-shell 究竟做了什么呢?

1. 脚本分析

在 Spark 安装目录的 bin 文件夹下可以找到 spark-shell，其中有代码清单 1-1 所示的一段脚本。

代码清单1-1 spark-shell脚本

```
function main() {
  if $cygwin; then
    stty -icanon min 1 -echo > /dev/null 2>&1
    export SPARK_SUBMIT_OPTS="$SPARK_SUBMIT_OPTS -Djline.terminal=unix"
    "${SPARK_HOME}"/bin/spark-submit --class org.apache.spark.repl.Main --name
        "Spark shell" "$@"
    stty icanon echo > /dev/null 2>&1
  else
    export SPARK_SUBMIT_OPTS
    "${SPARK_HOME}"/bin/spark-submit --class org.apache.spark.repl.Main --name
        "Spark shell" "$@"
  fi
}
```



我们看到脚本 spark-shell 里执行了 spark-submit 脚本，那么打开 spark-submit 脚本，发现代码清单 1-2 中所示的脚本。

代码清单1-2 spark-submit脚本

```
if [ -z "${SPARK_HOME}" ]; then
  source "$(dirname "$0")"/find-spark-home
fi
# disable randomized hash for string in Python 3.3+
export PYTHONHASHSEED=0
exec "${SPARK_HOME}"/bin/spark-class org.apache.spark.deploy.SparkSubmit "$@"
```

可以看到 spark-submit 中又执行了脚本 spark-class。打开脚本 spark-class，首先发现以下一段脚本：

```
# Find the java binary
if [ -n "${JAVA_HOME}" ]; then
  RUNNER="${JAVA_HOME}/bin/java"
else
  if [ "$(command -v java)" ]; then
    RUNNER="java"
  else
    echo "JAVA_HOME is not set" >&2
    exit 1
```

仅供非商业用途或交流学习使用


```
  fi
fi
```

上面的脚本是为了找到 Java 命令。在 spark-class 脚本中还会找到以下内容：

```
build_command() {
  "$RUNNER" -Xmx128m -cp "$LAUNCH_CLASSPATH" org.apache.spark.launcher.Main "$@"
  printf "%d\0" $?
}

CMD=()
while IFS= read -d '' -r ARG; do
  CMD+=("$ARG")
done < <(build_command "$@")
```

根据代码清单 1-2，脚本 spark-submit 在执行 spark-class 脚本时，给它增加了参数 SparkSubmit 。所以读到这里，应该知道 Spark 启动了以 SparkSubmit 为主类的 JVM 进程。

2. 远程监控

为便于在本地对 Spark 进程进行远程监控，在 spark-shell 脚本中找到以下配置：

```
SPARK_SUBMIT_OPTS="$SPARK_SUBMIT_OPTS -Dscala.usejavacp=true"
```

并追加以下 jmx 配置：

```
-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.port=10207
-Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.
    ssl=false
```



如果 Spark 安装在其他机器，那么在本地打开 Java VisualVM 后需要添加远程主机，如图 1-12 所示。

右击已添加的远程主机，添加 JMX 连接，如图 1-13 所示。



图 1-12　添加远程主机

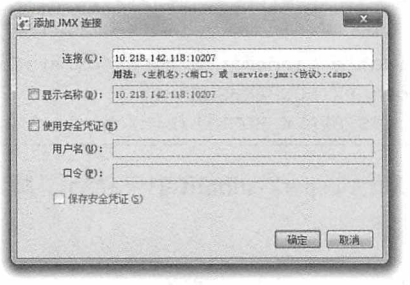


图 1-13　添加 JMX 连接

如果 Spark 安装在本地，那么打开 Java VisualVM 后就会在应用程序窗口中看到 org.apache.spark.deploy.SparkSubmit 进程，只需双击即可。

选择右侧的“线程”选项卡，选择 main 线程，然后点击“线程 Dump”按钮，如图 1-14 所示。

从线程 Dump 的内容中找到线程 main 的信息，如代码清单 1-3 所示。

仅供非商业用途或交流学习使用

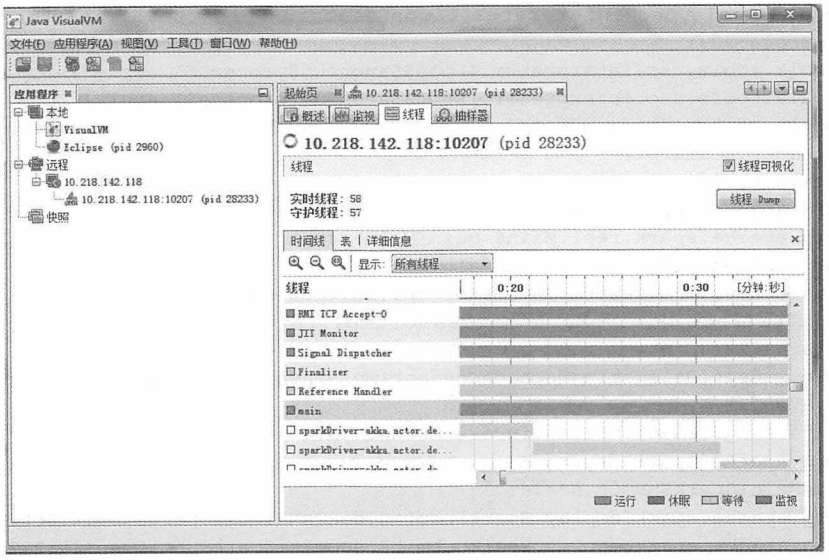


图 1-14 查看 Spark 线程

代码清单1-3 main线程的Dump信息

```
"main" #1 prio=5 os_prio=31 tid=0x00007fa012802000 nid=0x1303 runnable [0x0000000-
    10d11c000]
  java.lang.Thread.State: RUNNABLE
  at java.io.FileInputStream.read0(Native Method)
  at java.io.FileInputStream.read(FileInputStream.java:207)
  at jline.internal.NonBlockingInputStream.read(NonBlockingInputStream.java:169)
  - locked <0x00000007837a8ab8> (a jline.internal.NonBlockingInputStream)
  at jline.internal.NonBlockingInputStream.read(NonBlockingInputStream.java:137)
  at jline.internal.NonBlockingInputStream.read(NonBlockingInputStream.java:246)
  at jline.internal.InputStreamReader.read(InputStreamReader.java:261)
  - locked <0x00000007837a8ab8> (a jline.internal.NonBlockingInputStream)
  at jline.internal.InputStreamReader.read(InputStreamReader.java:198)
  - locked <0x00000007837a8ab8> (a jline.internal.NonBlockingInputStream)
  at jline.console.ConsoleReader.readCharacter(ConsoleReader.java:2145)
  at jline.console.ConsoleReader.readLine(ConsoleReader.java:2349)
  at jline.console.ConsoleReader.readLine(ConsoleReader.java:2269)
  at scala.tools.nsc.interpreter.jline.InteractiveReader.readOneLine(JLineReader.scala:57)
  at scala.tools.nsc.interpreter.InteractiveReader$$anonfun$readLine$2.
      apply(InteractiveReader.scala:37)
  at scala.tools.nsc.interpreter.InteractiveReader$$anonfun$readLine$2.
      apply(InteractiveReader.scala:37)
  at scala.tools.nsc.interpreter.InteractiveReader$.restartSysCalls(InteractiveR
      eader.scala:44)
  at scala.tools.nsc.interpreter.InteractiveReader$class.readLine(Interactive-
      Reader.scala:37)
  at scala.tools.nsc.interpreter.jline.InteractiveReader.readLine(JLineReader.scala:28)
  at scala.tools.nsc.interpreter.ILoop.readOneLine(ILoop.scala:404)
  at scala.tools.nsc.interpreter.ILoop.loop(ILoop.scala:413)
```



仅供非商业用途或交流学习使用


```
at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply$mcZ$sp(ILoop.
    scala:923)
at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply(ILoop.scala:909)
at scala.tools.nsc.interpreter.ILoop$$anonfun$process$1.apply(ILoop.scala:909)
at scala.reflect.internal.util.ScalaClassLoader$.savingContextLoader(ScalaClass
    Loader.scala:97)
at scala.tools.nsc.interpreter.ILoop.process(ILoop.scala:909)
at org.apache.spark.repl.Main$.doMain(Main.scala:68)
at org.apache.spark.repl.Main$.main(Main.scala:51)
at org.apache.spark.repl.Main.main(Main.scala)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
    java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
    java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.apache.spark.deploy.SparkSubmit$.org$apache$spark$deploy$SparkSubmit$$r
    unMain(SparkSubmit.scala:738)
at org.apache.spark.deploy.SparkSubmit$.doRunMain$1(SparkSubmit.scala:187)
at org.apache.spark.deploy.SparkSubmit$.submit(SparkSubmit.scala:212)
at org.apache.spark.deploy.SparkSubmit$.main(SparkSubmit.scala:126)
at org.apache.spark.deploy.SparkSubmit.main(SparkSubmit.scala)
```

从 main 线程的栈信息中可以看出程序的调用顺序：SparkSubmit.main → repl.Main → Iloop.process。



3. 源码分析

我们根据上面的线索，直接阅读 Iloop 的 process 方法的源码[⊖]，如代码清单 1-4 所示。

代码清单1-4　process的实现

```
def process(settings: Settings): Boolean = savingContextLoader {
  this.settings = settings
  createInterpreter()
  // sets in to some kind of reader depending on environmental cues
  in = in0.fold(chooseReader(settings))(r => SimpleReader(r, out, interactive =
      true))
  globalFuture = future {
    intp.initializeSynchronous()
    loopPostInit()
    !intp.reporter.hasErrors
  }
  loadFiles(settings)
    printWelcome()
  try loop() match {
    case LineResults.EOF => out print Properties.shellInterruptedString
    case _               =>
  }
  catch AbstractOrMissingHandler()
```

⊖ Iloop 是 Scala 语言自身的类库中的用于实现交互式 shell 的实现类，提供对 REPL（Read-eval-print-loop）的实现。

仅供非商业用途或交流学习使用


```
    finally closeInterpreter()
    true
  }
```

根据代码清单1-4，Iloop的process方法调用了loadFiles方法。Spark中的SparkILoop继承了Iloop并重写了loadFiles方法，其实现如下：

```
override def loadFiles(settings: Settings): Unit = {
  initializeSpark()
  super.loadFiles(settings)
}
```

根据上面展示的代码，loadFiles方法调用了SparkILoop的initializeSpark方法，initializeSpark的实现如代码清单1-5所示。

代码清单1-5 initializeSpark的实现

```
def initializeSpark() {
  intp.beQuietDuring {
    processLine("""
      @transient val spark = if (org.apache.spark.repl.Main.sparkSession != null) {
          org.apache.spark.repl.Main.sparkSession
        } else {
          org.apache.spark.repl.Main.createSparkSession()
        }
      @transient val sc = {
        val _sc = spark.sparkContext
        if (_sc.getConf.getBoolean("spark.ui.reverseProxy", false)) {
          val proxyUrl = _sc.getConf.get("spark.ui.reverseProxyUrl", null)
          if (proxyUrl != null) {
            println(s"Spark Context Web UI is available at ${proxyUrl}/proxy/${_
              sc.applicationId}")
          } else {
            println(s"Spark Context Web UI is available at Spark Master Public URL")
            }
          } else {
            _sc.uiWebUrl.foreach {
              webUrl => println(s"Spark context Web UI available at ${webUrl}")
            }
          }
          println("Spark context available as 'sc' " +
            s"(master = ${_sc.master}, app id = ${_sc.applicationId}).")
          println("Spark session available as 'spark'.")
          _sc
        }
        """)
      processLine("import org.apache.spark.SparkContext._")
      processLine("import spark.implicits._")
      processLine("import spark.sql")
      processLine("import org.apache.spark.sql.functions._")
      replayCommandStack = Nil // remove above commands from session history.
    }
  }
```

仅供非商业用途或交流学习使用

我们看到，initializeSpark 向交互式 shell 发送了一大串代码，Scala 的交互式 shell 将调用 org.apache.spark.repl.Main 的 createSparkSession 方法（见代码清单 1-6），创建 SparkSession。我们看到常量 spark 将持有 SparkSession 的引用，并且 sc 持有 SparkSession 内部初始化好的 SparkContext。所以我们才能够在 spark-shell 的交互式 shell 中使用 sc 和 spark。

代码清单1-6 createSparkSession的实现

```
def createSparkSession(): SparkSession = {
  val execUri = System.getenv("SPARK_EXECUTOR_URI")
  conf.setIfMissing("spark.app.name", "Spark shell")
  conf.set("spark.repl.class.outputDir", outputDir.getAbsolutePath())
  if (execUri != null) {
    conf.set("spark.executor.uri", execUri)
  }
  if (System.getenv("SPARK_HOME") != null) {
    conf.setSparkHome(System.getenv("SPARK_HOME"))
  }

  val builder = SparkSession.builder.config(conf)
  if (conf.get(CATALOG_IMPLEMENTATION.key, "hive").toLowerCase == "hive") {
    if (SparkSession.hiveClassesArePresent) {
      sparkSession = builder.enableHiveSupport().getOrCreate()
      logInfo("Created Spark session with Hive support")
    } else {
      builder.config(CATALOG_IMPLEMENTATION.key, "in-memory")
      sparkSession = builder.getOrCreate()
      logInfo("Created Spark session")
    }
  } else {
    sparkSession = builder.getOrCreate()
    logInfo("Created Spark session")
  }
  sparkContext = sparkSession.sparkContext
  sparkSession
}
```

根据代码清单 1-6 所示，createSparkSession 方法通过 SparkSession 的 API 创建 SparkSession 实例。本书将有关 SparkSession 等 API 的内容放在第 10 章进行讲解，初次接触 Spark 的读者现在只需要了解即可。

1.3 阅读环境准备

准备 Spark 阅读环境，同样需要一台好机器。笔者调试源码的机器内存是 8GB。源码阅读的前提是首先在 IDE 环境中打包、编译通过。常用的 IDE 有 IntelliJ IDEA 和 Eclipse，笔者选择用 Eclipse 编译和阅读 Spark 源码，原因有两个：一是由于使用多年，对它比较熟悉；二是社区中使用 Eclipse 编译 Spark 的资料太少，在这里可以做个补充。笔者在 Mac OS 系统中编译 Spark 源码，除了安装 JDK 和 Scala 外，还需要安装以下工具。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

1.3.1 安装 SBT

由于 Scala 使用 SBT 作为构建工具，所以需要下载 SBT。下载地址：http://www.scala-sbt.org/，下载安装包并安装，本书以 sbt-0.13.12.tgz 为例。

1）移动到选好的安装目录，例如：

```
mv sbt-0.13.12.tgz ~/install/
```

2）进入安装目录，执行以下命令：

```
chmod 755 sbt-0.13.12.tgz
tar -xzvf sbt-0.13.12.tgz
```

3）配置环境：

```
cd ~
vim .bash_profile
```

4）添加如下配置：

```
export SBT_HOME=$HOME/install/sbt
export PATH=$SBT_HOME/bin:$PATH
```

输入以下命令，使环境变量快速生效：

```
source .bash_profile
```

安装完毕后，使用 sbt about 命令查看，确认安装正常，如图 1-15 所示。

```
EBJ1266:sbt user$ sbt about
[info] Set current project to sbt (in build file:/Users/user/install/sbt/)
[info] This is sbt 0.13.12
[info] The current project is {file:/Users/user/install/sbt/}sbt 0.1-SNAPSHOT
[info] The current project is built against Scala 2.10.6
[info] Available Plugins: sbt.plugins.IvyPlugin, sbt.plugins.JvmPlugin, sbt.plugins.CorePlugin, sbt
[info] sbt, sbt plugins, and build definitions are using Scala 2.10.6
```

图 1-15 查看 SBT 安装是否正常

1.3.2 安装 Git

由于 Spark 源码使用 Git 作为版本控制工具，所以需要下载 Git 的客户端工具。下载地址：https://git-scm.com，下载最新的版本并安装。

安装完毕后可使用 git --version 命令来查看安装是否正常，如图 1-16 所示。

```
EBJ1266:sbt user$ git --version
git version 2.7.1
```

图 1-16 查看 Git 是否安装成功

1.3.3 安装 Eclipse Scala IDE 插件

Eclipse 通过强大的插件方式支持各种 IDE 工具的集成，要在 Eclipse 中编译、调

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

试、运行 Scala 程序，就需要安装 Eclipse Scala IDE 插件。下载地址：http://scala-ide.org/download/current.html。

由于笔者本地的 Eclipse 版本是 Eclipse Mars.2 Release（4.5.2），所以选择安装插件 http://download.scala-ide.org/sdk/lithium/e44/scala211/stable/site，如图 1-17 所示。

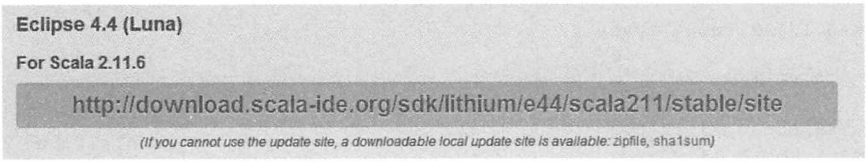


图 1-17　Eclipse Scala IDE 插件安装地址

在 Eclipse 中选择 Help 菜单，然后选择 Install New Software 选项，打开 Install 对话框，如图 1-18 所示。

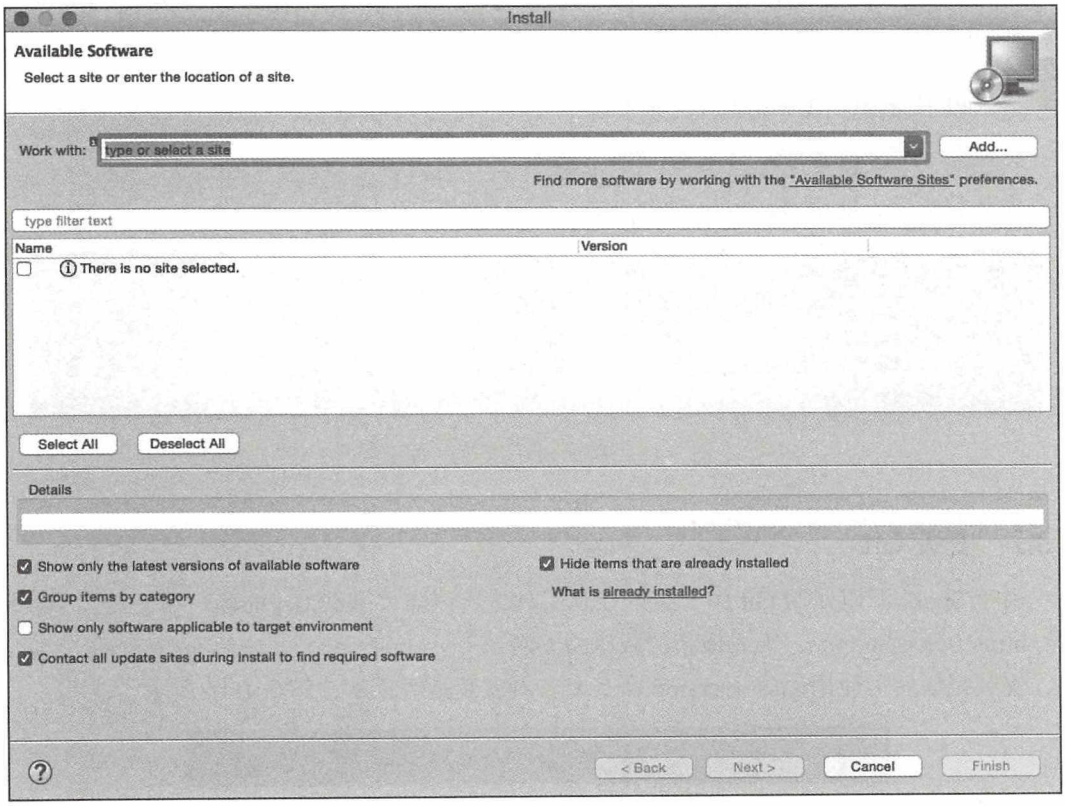


图 1-18　打开 Install 对话框

点击 Add 按钮，打开 Add Repository 对话框，输入插件地址，如图 1-19 所示。

全选插件的内容，完成安装，如图 1-20 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

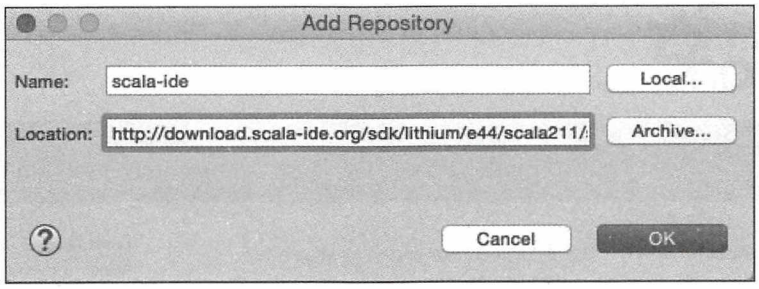


图 1-19 添加 Scala IDE 插件地址

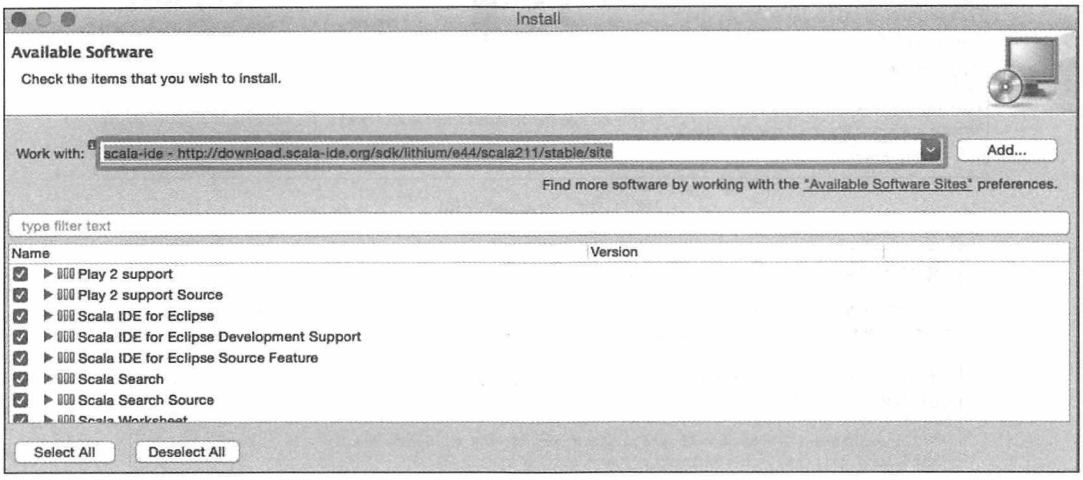


图 1-20 完成安装 Scala IDE 插件

1.4 Spark 源码编译与调试

1. 下载 Spark 源码

首先，访问 Spark 官网 http://spark.apache.org/，如图 1-21 所示。

点击 Download Spark 按钮，在下一个页面找到 Git 地址，如图 1-22 所示。

笔者在当前用户目录下创建 Source 文件夹作为放置 Spark 源码的地方，进入此文件夹并输入 git clone git://github.com/apache/spark.git 命令将源码下载到本地，如图 1-23 所示。

2. 构建 Scala 应用

进入 Spark 根目录，执行 sbt 命令。然后会下载和解析很多 Jar 包，要等待一段时间，笔者大概花费了一个多小时才执行完，如图 1-24 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

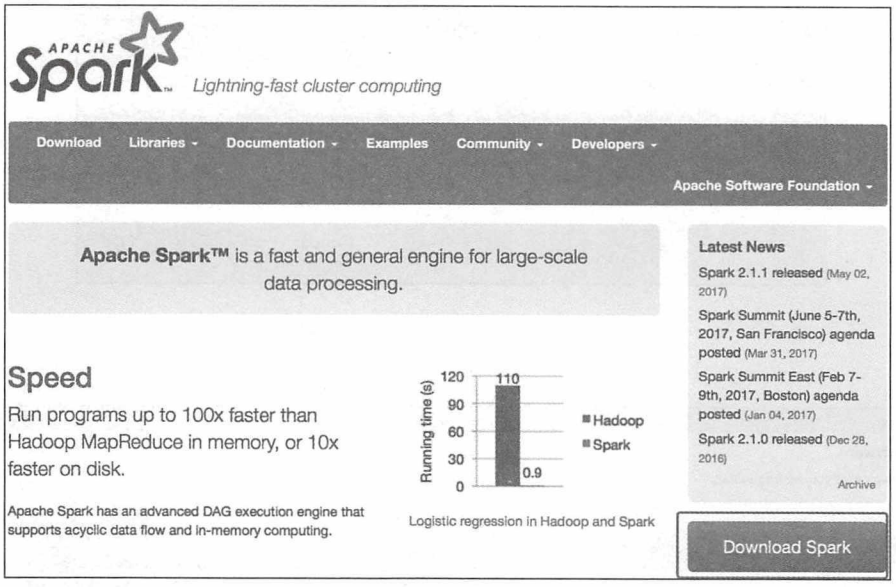


图 1-21 Spark 官网

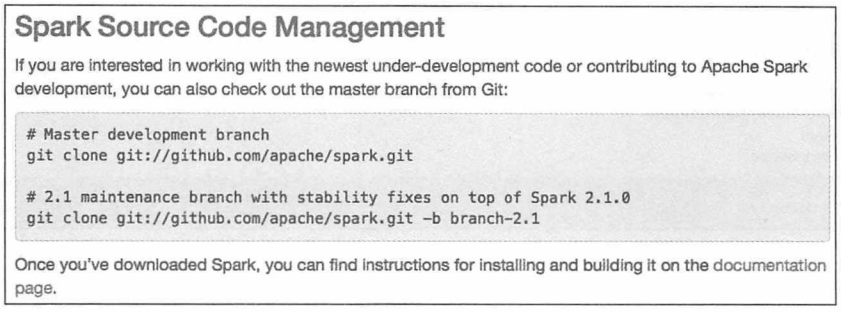


图 1-22 Spark 官方 Git 地址

```
EBJ1266:~ user$ cd Source/
EBJ1266:Source user$ git clone git://github.com/apache/spark.git
Cloning into 'spark'...
remote: Counting objects: 479752, done.
remote: Compressing objects: 100% (110/110), done.
Receiving objects:   0% (1849/479752), 1.11 MiB | 23.00 KiB/s
```

图 1-23 下载 Spark 源码

```
EBJ1266:spark user$ sbt
[info] Loading project definition from /Users/user/resGit/spark/project
[info] Set current project to spark-parent (in build file:/Users/user/resGit/spark/)
>
```

图 1-24 构建 Scala 应用

从图 1-24 中可以看出，sbt 构建完毕时会出现提示符“>”。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

3. 使用 sbt 生成 Eclipse 工程文件

在 sbt 命令出现提示符“>”后，输入 eclipse 命令，开始生成 Eclipse 工程文件，也需要花费很长的时间，笔者大致花费了 40 分钟。完成时的状况如图 1-25 所示。

```
[info] Done packaging.
[info] Compiling 4 Scala sources to /Users/user/resGit/spark/repl/target/scala-2.11/classes...
[info] Resolving jline#jline;2.12.1 ...
[info] downloading https://repo1.maven.org/maven2/com/github/scopt/scopt_2.11/3.3.0/scopt_2.11-3.3.0.
[warn] there were 16 deprecation warnings; re-run with -deprecation for details
[warn] one warning found
[info]  [SUCCESSFUL ] com.github.scopt#scopt_2.11;3.3.0!scopt_2.11.jar (5468ms)
[info] Done updating.
[info] Resolving org.apache.hadoop#hadoop-client;2.2.0 ...
[info] Packaging /Users/user/resGit/spark/sql/hive/target/scala-2.11/spark-hive_2.11-2.0.3-SNAPSHOT.j
[info] Resolving org.apache.hadoop#hadoop-mapreduce-client-common;2.2.0 ...
[info] Done packaging.
[info] Resolving org.apache.hadoop#hadoop-yarn-common;2.2.0 ...
[info] Compiling 179 Scala sources and 114 Java sources to /Users/user/resGit/spark/examples/target/s
[info] Resolving com.typesafe.genjavadoc#genjavadoc-plugin_2.11.8;0.10 ...
[info] Done updating.
[info] Packaging /Users/user/resGit/spark/repl/target/scala-2.11/spark-repl_2.11-2.0.3-SNAPSHOT.jar ...
[info] Done packaging.
[warn] /Users/user/resGit/spark/examples/src/main/scala/org/apache/spark/examples/mllib/AbstractParams
ypeApi is deprecated: Use `decls` instead
[warn]     val allAccessors = tpe.declarations.collect {
[warn]                                ^
[warn] one warning found
[warn] /Users/user/resGit/spark/examples/src/main/scala/org/apache/spark/examples/mllib/AbstractParams
ypeApi is deprecated: Use `decls` instead
[warn]     val allAccessors = tpe.declarations.collect {
[warn]
[success] Total time: 4935 s, completed 2017-2-14 15:11:59
[3]+  Killed: 9               sbt compile
```

图 1-25　sbt 编译过程

现在我们查看 Spark 下的子文件夹，发现其中都生成了 .project 和 .classpath 文件。比如 mllib 项目下就生成了 .project 和 .classpath 文件，如图 1-26 所示。

4. 编译 Spark 源码

由于 Spark 使用 Maven 作为项目管理工具，所以需要将 Spark 项目作为 Maven 项目导入到 Eclipse 中，如图 1-27 所示。

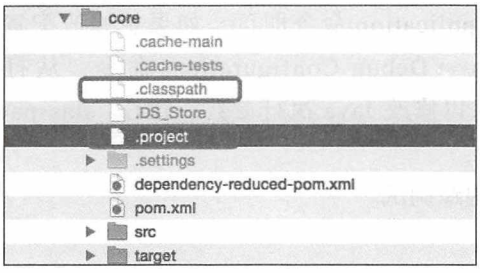


图 1-26　sbt 生成的项目文件

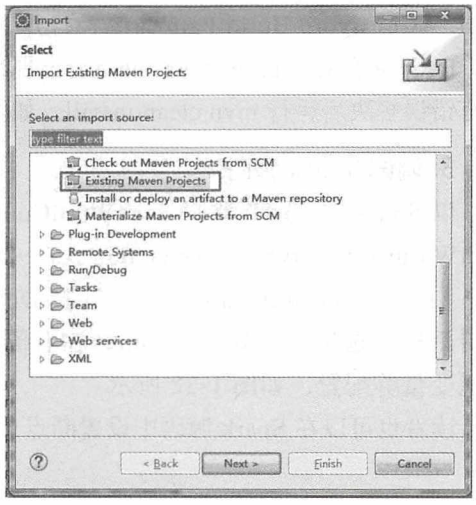


图 1-27　导入 Maven 项目

書籍是人类进步的阶梯

仅供非商业用途或交流学习使用

点击 Next 按钮进入下一个对话框，如图 1-28 所示。

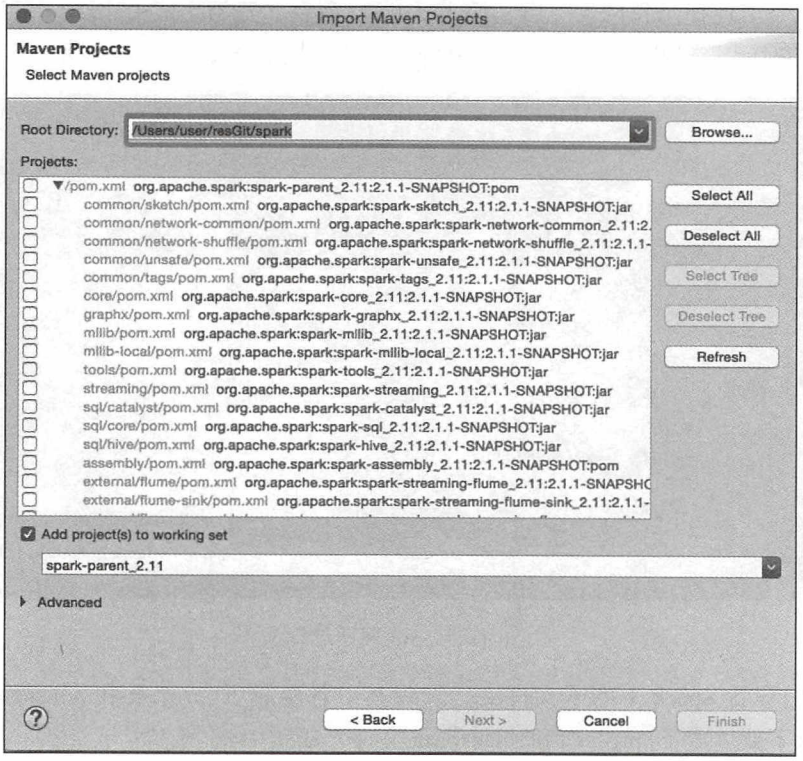


图 1-28　选择 Maven 项目

全选所有项目，点击 finish 按钮，这样就完成了导入，如图 1-29 所示。

导入完成后，需要设置每个子项目的 Build Path。右击每个项目，选择 Build Path → Configure Build Path 命令，打开 Java Build Path 对话框，如图 1-30 所示。

Eclipse 在对项目编译时，可能会出现很多错误，只要仔细分析报错原因就能一一排除。所有错误解决后运行 mvn clean install，如图 1-31 所示。

5. 调试 Spark 源码

以 Spark 源码自带的 JavaWordCount 为例，介绍如何调试 Spark 源码。右击 JavaWordCount.java，选择 Debug As → Java Application 命令即可。如果想修改配置参数，右击 JavaWordCount.java，选择 Debug As → Debug Configurations 命令，从打开的对话框中选择 JavaWordCount，在右侧区域可以修改 Java 执行参数、JRE、classpath、环境变量等配置，如图 1-32 所示。

读者也可以在 Spark 源码中设置断点，进行跟踪调试。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

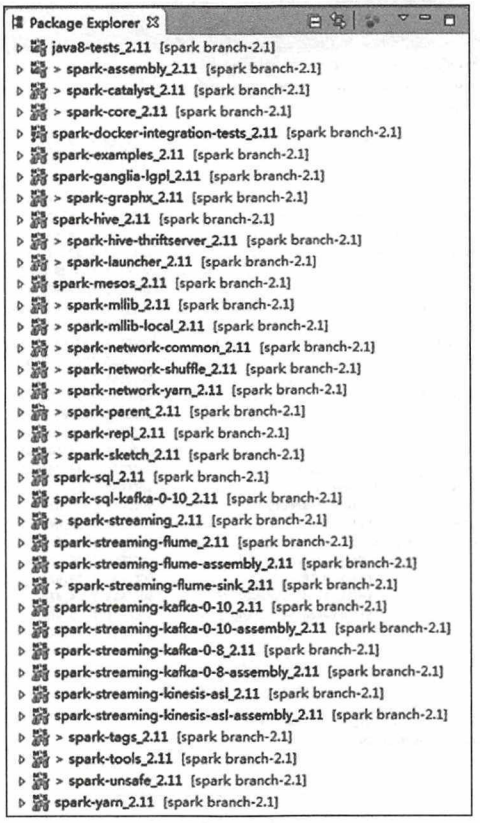


图 1-29 导入完成的项目

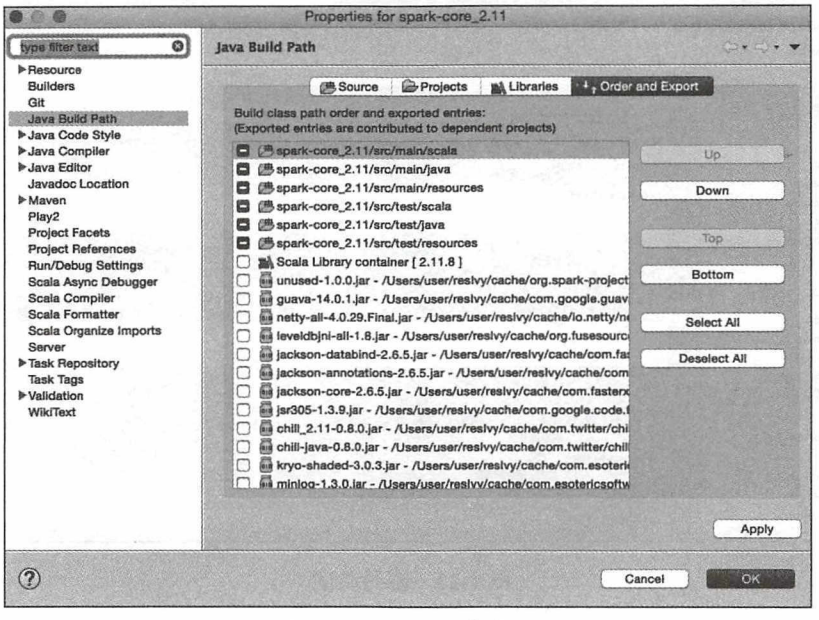


图 1-30 Java 构建路径

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

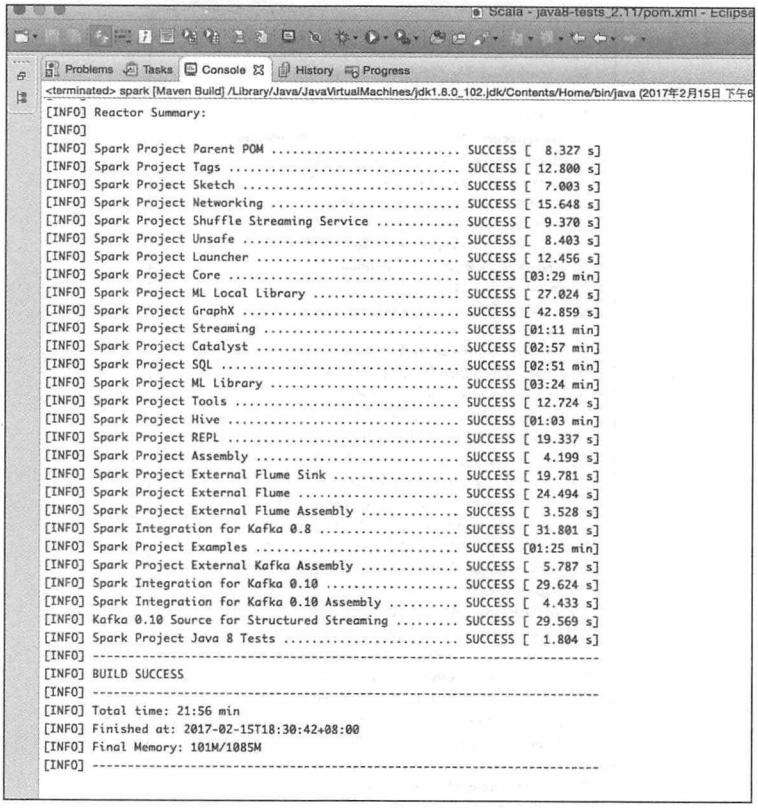


图 1-31 编译成功

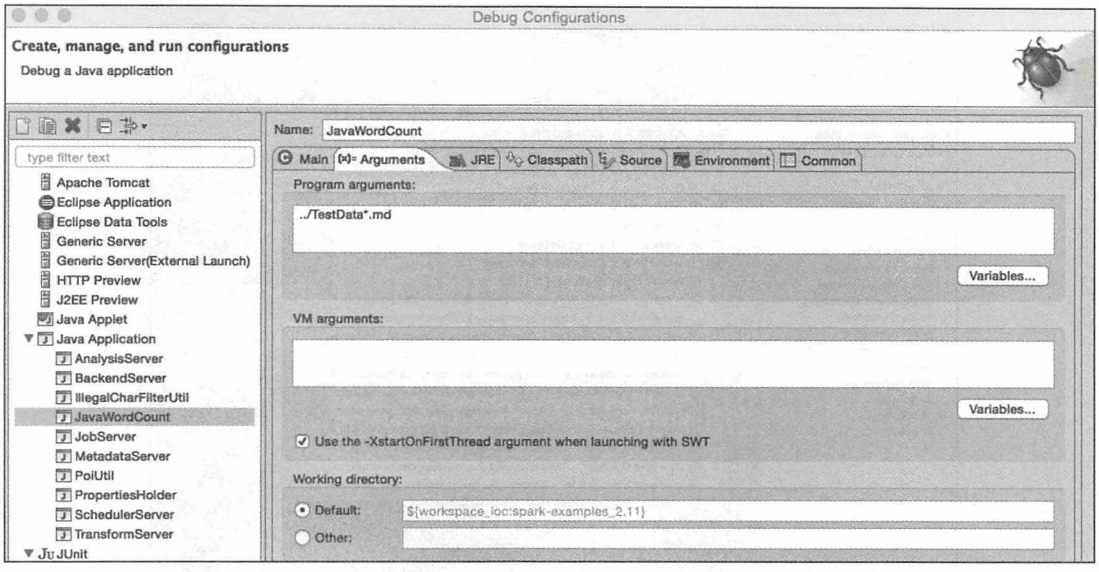


图 1-32 源码调试

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

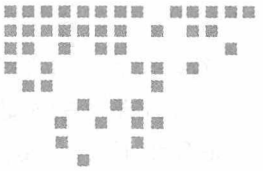
1.5 小结

本章通过引导大家在 Linux 操作系统下搭建基本的执行环境，并且介绍 spark-shell 等脚本的执行，帮助读者由浅入深地进行 Spark 源码的学习。由于笔者的多数开发工作都在 Mac 系统下，所以本书介绍的搭建源码阅读环境是在 Mac 环境下。Eclipse 有广大的用户群，即便是一些开始使用 IntelliJ 的用户，对 Eclipse 也不陌生，所以本章选择 Eclipse 作为导入、编译、阅读、调试 Spark 源码的工具，希望能降低读者的学习门槛。



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用



Chapter 2 第 2 章

设计理念与基本架构

“若夫乘天地之正，而御六气之辩，以游无穷者，彼且恶乎待哉？”

——《庄子·逍遥游》



本章导读

随着近十年互联网的迅猛发展，越来越多的人融入了互联网——利用搜索引擎查询词条或问题；社交圈子从现实搬到了 Facebook、Twitter、微信等社交平台上；女孩子们现在少了逛街，多了在各大电商平台上的购买；喜欢棋牌的人能够在对战平台上找到世界各地的玩家对弈。在国内随着网民数量的持续增加，互联网公司的数据在体量、产生速度、多样性等方面呈现出巨大的变化。

互联网产生的数据相比传统软件产生的数据而言，有着数据挖掘的巨大潜力。通过对数据的挖掘，可以统计出 PV、UV，计算出不同设备与注册率、促销与下单率之间的关系，甚至构建热点分析、人群画像等算法模型，产生一系列报表、图形、离线统计、实时计算的产品。互联网公司如果能有效利用这些数据，将对决策和战略发展起到至关重要的作用。

在大数据的大势之下，Hadoop、Spark、Flink、Storm、Dremel、Impala、Tez 等一系列大数据技术如雨后春笋般不断涌现。工程师们正在使用这些工具摸索前行。

上一章介绍了 Spark 运行环境和源码阅读环境的搭建，为读者学习 Spark 做好准备。本章首先从 Spark 产生的背景开始，介绍 Spark 的主要特点、基本概念、版本变迁，然后简要说明 Spark 的主要模块和编程模型。最后从 Spark 的设计理念和基本架构入手，使读者能够对 Spark 有一个宏观的认识，为之后的内容打好基础。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

2.1　初识 Spark

Spark 是一个通用的并行计算框架，由加州伯克利大学（UC Berkeley）的 AMP 实验室开发于 2009 年，并于 2010 年开源，2013 年成长为 Apache 旗下在大数据领域最活跃的开源项目之一。

Spark 目前已经走过了 0.x 和 1.x 两个时代，现在正在 2.x 时代稳步发展。Spark 从 2012 年 10 月 15 日发布 Spark 0.6 到 2016 年 1 月 4 日发布 Spark 1.6 只经过了三年时间，那时候差不多每个月都会有新的版本发布，平均每个季度会发布一个新的二级版本。

自从 2016 年 7 月发布了 Spark 2.0.0 版本以来，只在当年 12 月发布了 Spark 2.1.0 版本，到目前为止还没有新的二级版本发布。Spark 发布新版本的节奏明显慢了下来，当然这也跟 Spark 团队过于激进的决策（比如很多 API 不能向前兼容）有关。

Spark 也是基于 map reduce 算法模型实现的分布式计算框架，拥有 Hadoop MapReduce 所具有的优点，并且解决了 Hadoop MapReduce 中的诸多缺陷。

2.1.1　Hadoop MRv1 的局限

早在 Hadoop 1.0 版本，当时采用的是 MRv1 版本的 MapReduce 编程模型。MRv1 版本的实现都封装在 org.apache.hadoop.mapred 包中，MRv1 的 Map 和 Reduce 是通过接口实现的。MRv1 包括以下三个部分。



- 运行时环境（JobTracker 和 TaskTracker）。
- 编程模型（MapReduce）。
- 数据处理引擎（Map 任务和 Reduce 任务）。

MRv1 存在以下不足。

- **可扩展性差**：在运行时，JobTracker 既负责资源管理，又负责任务调度，当集群繁忙时，JobTracker 很容易成为瓶颈，最终导致它的可扩展性问题。
- **可用性差**：采用了单节点的 Master，没有备用 Master 及选举操作，这导致一旦 Master 出现故障，整个集群将不可用。
- **资源利用率低**：TaskTracker 使用 slot 等量划分本节点上的资源量。slot 代表计算资源（CPU、内存等）。一个 Task 获取到一个 slot 后才有机会运行，Hadoop 调度器负责将各个 TaskTracker 上的空闲 slot 分配给 Task 使用。一些 Task 并不能充分利用 slot，而其他 Task 也无法使用这些空闲的资源。slot 分为 Map slot 和 Reduce slot 两种，分别供 MapTask 和 Reduce Task 使用。有时会因为作业刚刚启动等原因导致 MapTask 很多，而 Reduce Task 任务还没有调度的情况，这时 Reduce slot 也会被闲置。
- **不能支持多种 MapReduce 框架**：无法通过可插拔方式将自身的 MapReduce 框架替换为其他实现，如 Spark、Storm 等。

仅供非商业用途或交流学习使用

MRv1 的示意如图 2-1 所示。

Apache 为了解决以上问题，对 Hadoop 进行升级改造，于是 MRv2 最终诞生了。在 MRv2 中，重用了 MRv1 中的编程模型和数据处理引擎。但是运行时环境被重构了。JobTracker 被拆分成通用的资源调度平台 ResourceManager（简称 RM），节点管理器 NodeManager 和负责各个计算框架的任务调度模型 ApplicationMaster（简称 AM）。ResourceManager 依然负责对整个集群的资源管理，但是在任务资源的调度方面只负责将资源封装为 Container 分配给 ApplicationMaster 的一级调度，二级调度的细节将交给 ApplicationMaster 去完成，这大大减轻了 ResourceManager 的压力，使得 ResourceManager 更加轻量。NodeManager 负责对单个节点的资源管理，并将资源信息、Container 运行状态、健康状况等信息上报给 ResourceManager。ResourceManager 为了保证 Container 的利用率，会监控 Container，如果 Container 未在有限的时间内使用，ResourceManager 将命令 NodeManager “杀死” Container，以便将资源分配给其他任务。MRv2 中 MapReduce 的核心不再是 MapReduce 框架，而是 YARN。在以 YARN 为核心的 MRv2 中，MapReduce 框架是可插拔的，完全可以替换为其他 MapReduce 实现，比如 Spark、Storm 等。MRv2 的示意如图 2-2 所示。



图 2-1 MRv1 示意图[⊖]

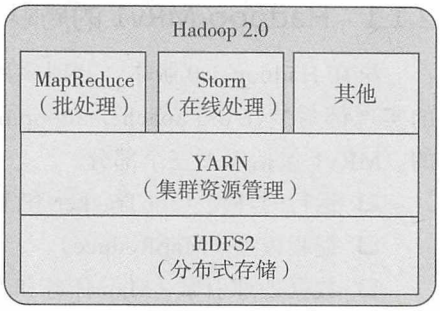


图 2-2 MRv2 示意图

Hadoop MRv2 虽然解决了 MRv1 中的一些问题，但是由于对 HDFS 的频繁操作（包括计算结果持久化、数据备份、资源下载及 Shuffle 等），导致磁盘 I/O 成为系统性能的瓶颈，因此只适用于离线数据处理或批处理，而不能支持对迭代式、流式数据的处理。

2.1.2 Spark 的特点

Spark 看到 MRv1 的问题，对 MapReduce 做了大量优化，总结如下。

1）**减少磁盘 I/O**：随着实时大数据应用越来越多，Hadoop 作为离线的高吞吐、低响应框架已不能满足这类需求。Hadoop MapReduce 的 map 端将中间输出和结果存储在磁盘中，reduce 端又需要从磁盘读写中间结果，势必造成磁盘 I/O 成为瓶颈。Spark 允许将 map 端的中间输出和结果存储在内存中，reduce 端在拉取中间结果时避免了大量的磁盘 I/O。Hadoop YARN 中的 ApplicationMaster 申请到 Container 后，具体任务需要利用 NodeManager

⊖ 图 2-1 和图 2-2 都来源自 http://blog.chinaunix.net/uid-28311809-id-4383551.html。

仅供非商业用途或交流学习使用

从 HDFS 的不同节点下载任务所需的资源（如 Jar 包），这也增加了磁盘 I/O。Spark 将应用程序上传的资源文件缓冲到 Driver 本地文件服务的内存中，当 Executor 执行任务时直接从 Driver 的内存中读取，也节省了大量的磁盘 I/O。

2）**增加并行度**：由于将中间结果写到磁盘与从磁盘读取中间结果属于不同的环节，Hadoop 将它们简单地通过串行执行衔接起来。Spark 把不同的环节抽象为 Stage，允许多个 Stage 既可以串行执行，又可以并行执行。

3）**避免重新计算**：当 Stage 中某个分区的 Task 执行失败后，会重新对此 Stage 调度，但在重新调度的时候会过滤已经执行成功的分区任务，所以不会造成重复计算和资源浪费。

4）**可选的 Shuffle 排序**：Hadoop MapReduce 在 Shuffle 之前有着固定的排序操作，而 Spark 则可以根据不同场景选择在 map 端排序还是 reduce 端排序。

5）**灵活的内存管理策略**：Spark 将内存分为堆上的存储内存、堆外的存储内存、堆上的执行内存、堆外的执行内存 4 个部分。Spark 既提供了执行内存和存储内存之间固定边界的实现，又提供了执行内存和存储内存之间“软”边界的实现。Spark 默认使用“软”边界的实现，执行内存或存储内存中的任意一方在资源不足时都可以借用另一方的内存，最大限度地提高资源的利用率，减少对资源的浪费。Spark 由于对内存使用的偏好，内存资源的多寡和使用率就显得尤为重要，为此 Spark 的内存管理器提供的 Tungsten 实现了一种与操作系统的内存 Page 非常相似的数据结构，用于直接操作操作系统内存，节省了创建的 Java 对象在堆中占用的内存，使得 Spark 对内存的使用效率更加接近硬件。Spark 会给每个 Task 分配一个配套的任务内存管理器，对 Task 粒度的内存进行管理。Task 的内存可以被多个内部的消费者消费，任务内存管理器对每个消费者进行 Task 内存的分配与管理，因此 Spark 对内存有着更细粒度的管理。

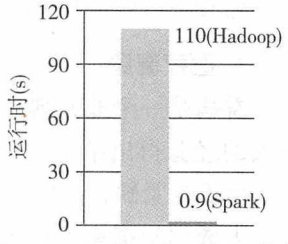


图 2-3　Hadoop 与 Spark 执行逻辑回归时间比较

基于以上所列举的优化，Spark 官网声称其性能比 Hadoop 快 100 倍，如图 2-3 所示。即便是内存不足需要磁盘 I/O 时，其速度也是 Hadoop 的 10 倍以上。

Spark 还有其他一些特点。

1）**检查点支持**：Spark 的 RDD 之间维护了血缘关系（lineage），一旦某个 RDD 失败了，则可以由父 RDD 重建。虽然 lineage 可用于错误后 RDD 的恢复，但对于很长的 lineage 来说，恢复过程非常耗时。如果应用启用了检查点，那么在 Stage 中的 Task 都执行成功后，SparkContext 将把 RDD 计算的结果保存到检查点，这样当某个 RDD 执行失败后，再由父 RDD 重建时就不需要重新计算，而直接从检查点恢复数据。

2）**易于使用**。Spark 现在支持 Java、Scala、Python 和 R 等语言编写应用程序，大大降低了使用者的门槛。除此之外，还自带了 80 多个高等级操作符，允许在 Scala、Python、R 的 shell 中进行交互式查询。

3）**支持交互式**：Spark 使用 Scala 开发，并借助于 Scala 类库中的 Iloop 实现交互式

仅供非商业用途或交流学习使用

shell，提供对 REPL（Read-eval-print-loop）的实现。

4）**支持 SQL 查询**。在数据查询方面，Spark 支持 SQL 及 Hive SQL，这极大地方便了传统 SQL 开发和数据仓库的使用者。

5）**支持流式计算**：与 MapReduce 只能处理离线数据相比，Spark 还支持实时的流计算。Spark 依赖 Spark Streaming 对数据进行实时的处理，其流式处理能力还要强于 Storm。

6）**可用性高**。Spark 自身实现了 Standalone 部署模式，此模式下的 Master 可以有多个，解决了单点故障问题。Spark 也完全支持使用外部的部署模式，比如 YARN、Mesos、EC2 等。

7）**丰富的数据源支持**：Spark 除了可以访问操作系统自身的文件系统和 HDFS 之外，还可以访问 Kafka、Socket、Cassandra、HBase、Hive、Alluxio（Tachyon）及任何 Hadoop 的数据源。这极大地方便了已经使用 HDFS、HBase 的用户顺利迁移到 Spark。

8）**丰富的文件格式支持**：Spark 支持文本文件格式、CSV 文件格式、JSON 文件格式、ORC 文件格式、Parquet 文件格式、Libsvm 文件格式，也有利于 Spark 与其他数据处理平台的对接。

2.1.3 Spark 使用场景

Hadoop 常用于解决高吞吐、批量处理的业务场景，例如，对浏览量的离线统计。如果需要实时查看浏览量统计信息，Hadoop 显然不符合这样的要求。Spark 通过内存计算能力极大地提高了大数据处理速度，满足了以上场景的需要。此外，Spark 还支持交互式查询、SQL 查询、流式计算、图计算、机器学习等。通过对 Java、Python、Scala、R 等语言的支持，极大地方便了用户的使用。



笔者就目前所知道的 Spark 应用场景进行介绍。

1. 医疗健康

看病是一个非常典型的分析过程——医生根据患者的一些征兆、检验结果，结合医生本人的经验得出结论，最后给出相应的治疗方案。各地区医疗水平参差不齐，特别是高水平医生更为紧缺，好医院的地区分布很不均衡。

大数据根据患者的患病征兆、检验报告，通过病理分析模型找出病因并给出具体的治疗方案。即便是医疗水平落后的地区，只需要输入患者的患病征兆和病例数据，即可体验高水平医师的服务。通过 Spark 从海量数据中实时计算出病因，各个地区的医疗水平和效率将获得大幅度提升，同时也能很好地降低因为医生水平而导致误诊的概率。

实施医疗健康的必然措施是监测和预测。通过监测不断更新整个医疗基础库的知识，并通过医疗健康模型预测出疾病易发的地区和人群。

2. 电商

通过对用户的消费习惯、季节、产品使用周期等数据的收集，建立算法模型来判断消费者未来一个月、几个月甚至一年的消费需求（不是简单地根据你已经消费的产品，显示推

仅供非商业用途或交流学习使用

荐广告位），进而提高订单转化率。

在市场营销方面，通过给买家打标签，构建人群画像，进而针对不同的人群，精准投放广告、红包或优惠券。

3. 安全领域

面对日益复杂的网络安全，通过检测和数据分析区分出不同的安全类型，并针对不同的安全类型，实施不同的防御和打击措施。

- 端安全：使用安全卫士、云查杀对经过大数据分析得到的病毒、木马等进行防御。
- 电商安全：反刷单、反欺诈、合规。
- 金融安全：风险控制。
- 企业安全：反入侵。
- 国家安全：舆情监测，打击罪犯。

4. 金融领域

构建金融云，通过对巨量的计量数据进行收集，然后通过 Spark 实时处理分析，利用低延迟的数据处理能力，应对急迫的业务需求和数据增长。

量化投资，收集大宗商品的价格，以及黄金，石油等各种数据，分析黄金、股票等指数趋势，支持投资决策。

除了以上领域外，Spark 在搜索引擎、生态圈异常检测、生物计算等诸多领域都有广泛的应用场景。



2.2　Spark 基础知识

Spark 在 2009 年诞生于 UC Berkeley 的 AMP 实验室，2010 年正式对外开源。经过 5 年多的发展，Spark 目前的最新版本是 2.2.0。本书在写作之初选择了 Spark 2.1.0 作为分析的主要版本，相信绝大多数内容依然会满足读者朋友的需要。本节将从 Spark 的基本概念和开发语言两个角度来介绍 Spark 的基础知识。

1. 基本概念

要想对 Spark 有整体性的了解，推荐读者阅读 Matei Zaharia 的 Spark 论文。此处笔者先介绍 Spark 中的一些概念。

- RDD（resillient distributed dataset）：弹性分布式数据集。Spark 应用程序通过使用 Spark 的转换 API，可以将 RDD 封装为一系列具有血缘关系的 RDD，也就是 DAG。只有通过 Spark 的动作 API 才会将 RDD 及其 DAG 提交到 DAGScheduler。RDD 的祖先一定是一个跟数据源相关的 RDD，负责从数据源迭代读取数据。
- DAG（Directed Acycle graph）：有向无环图。在图论中，如果一个有向图无法从某个顶点出发经过若干条边回到该点，则这个图是一个有向无环图（DAG 图）。Spark

仅供非商业用途或交流学习使用

使用 DAG 来反映各 RDD 之间的依赖或血缘关系。

- Partition：数据分区，即一个 RDD 的数据可以划分为多少个分区。Spark 根据 Partition 的数量来确定 Task 的数量。
- NarrowDependency：窄依赖，即子 RDD 依赖于父 RDD 中固定的 Partition。NarrowDependency 分为 OneToOneDependency 和 RangeDependency 两种。
- ShuffleDependency：Shuffle 依赖，也称为宽依赖，即子 RDD 对父 RDD 中的所有 Partition 都可能产生依赖。子 RDD 对父 RDD 各个 Partition 的依赖将取决于分区计算器（Partitioner）的算法。
- Job：用户提交的作业。当 RDD 及其 DAG 被提交给 DAGScheduler 调度后，DAGScheduler 会将所有 RDD 中的转换及动作视为一个 Job。一个 Job 由一到多个 Task 组成。
- Stage：Job 的执行阶段。DAGScheduler 按照 ShuffleDependency 作为 Stage 的划分节点对 RDD 的 DAG 进行 Stage 划分（上游的 Stage 将为 ShuffleMapStage）。因此一个 Job 可能被划分为一到多个 Stage。Stage 分为 ShuffleMapStage 和 ResultStage 两种。
- Task：具体执行任务。一个 Job 在每个 Stage 内都会按照 RDD 的 Partition 数量，创建多个 Task。Task 分为 ShuffleMapTask 和 ResultTask 两种。ShuffleMapStage 中的 Task 为 ShuffleMapTask，而 ResultStage 中的 Task 为 ResultTask。ShuffleMapTask 和 ResultTask 类似于 Hadoop 中的 Map 任务和 Reduce 任务。
- Shuffle：Shuffle 是所有 MapReduce 计算框架的核心执行阶段，Shuffle 用于打通 map 任务（在 Spark 中就是 ShuffleMapTask）的输出与 reduce 任务（在 Spark 中就是 ResultTask）的输入，map 任务的中间输出结果按照指定的分区策略（例如，按照 key 值哈希）分配给处理某一个分区的 reduce 任务。

2. 开发语言

目前越来越多的语言可以运行在 Java 虚拟机上，Java 平台上的多语言混合编程正成为一种潮流。在混合编程模式下可以充分利用每种语言的特点和优势，以便更好地完成功能。Spark 同时选择了 Scala 和 Java 作为开发语言，也是为了充分利用二者各自的优势。表 2-1 所示是对这两种语言进行的比较。



注意 虽然 Actor 是 Scala 语言最初进行推广时最吸引人的特性之一，但是随着 Akka 更加强大的 Actor 类库的出现，Scala 已经在官方网站宣布废弃 Scala 自身的 Actor 编程模型，转而全面拥抱 Akka 提供的 Actor 编程模型。与此同时，从 Spark 2.0.0 版本开始，Spark 却放弃了使用 Akka，转而使用 Netty 实现了自己的 RPC 框架。遥想当年 Scala“鼓吹”Actor 编程模型优于 Java 的同步编程模型时，又有谁会想到如今这种场面呢？



仅供非商业用途或交流学习使用

表 2-1　Scala 与 Java 的比较

	Scala	Java
语言类型	面向函数为主，兼有面向对象	面向对象（Java 8 也增加了 Lambda 函数编程）
简洁性	非常简洁	不简洁
类型推断	丰富的类型推断，例如，深度和链式的类型推断、duck type、隐式类型转换等，但也因此增加了编译时长	少量的类型推断
可读性	一般，丰富的语法糖导致的各种奇幻用法，例如，方法签名、隐式转换	好
学习成本	较高	一般
语言特性	非常丰富的语法糖和更现代的语言特性，例如，Option、模式匹配、使用空格的方法调用	丰富
并发编程	使用 Actor 的消息模型	使用阻塞、锁、阻塞队列等

Scala 作为函数式编程的代表，天生适合并行运行，如果用 Java 语言实现相同的功能，会显得非常臃肿。很多介绍 Spark 的新闻或文章经常以 Spark 内核代码行数少或 API 精炼等内容作为宣传的“法器”，这应该也是选择 Scala 的原因之一。另一方面，由于函数式编程更接近计算机思维，因此便于通过算法从大数据中建模，这也更符合 Spark 作为大数据框架的理念。



由于 Java 适合服务器、中间件开发，所以 Spark 使用 Java 更多的是开发底层的基础设施或中间件。

2.3　Spark 基本设计思想

Spark 是一个功能丰富的大数据计算平台，与所有的大型系统一样，Spark 从设计到开发，也根据功能的不同进行模块的拆分。Spark 包含很多子模块，理解这些模块有助于我们掌握 Spark 的设计思想。Spark 的模块按照重要程度可分为核心功能和扩展功能。核心功能是 Spark 设计理念的核心实现，也是 Spark 陆续加入新功能的基础。在核心功能之上，通过不断地将丰富的扩展功能持续集成到 Spark 中，使得 Spark 满足更多市场、应用和用户的需求，促使 Spark 生态圈更加繁荣。

Spark 在编程模型方面没有采用 Hadoop 的 MapReduce 编程模型，而是封装了丰富的转换与执行的 API。开发人员可以充分利用这些 API 编写自己的 Spark 应用程序，而不用拘泥于如何套用 MapReduce 编程模型中的 Mapper 或 Reducer。Spark 提供的转换 API 将会在底层将数据封装为 RDD，并对这些 RDD 进行转换、构建血缘、构建 DAG、分片划分等处理，这些过程都是透明的，使得开发人员能更加专注于自己的业务实现。

下面就 Spark 的模块设计与模型设计，进行更详细的介绍。

仅供非商业用途或交流学习使用

2.3.1 Spark 模块设计

整个 Spark 主要由 Spark Coke、Spark SQL、Spark Streaming、GraphX、MLlib 组成，而后四项的能力都是建立在核心引擎之上，如图 2-4 所示。

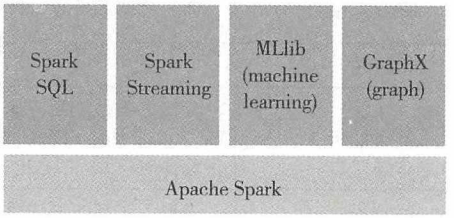


图 2-4 Spark 各模块的依赖关系

我们先来看看 Spark 的核心功能 Spark Core 的介绍。

1. Spark 核心功能

Spark Core 中提供了 Spark 最基础与最核心的功能，主要包括以下几项。

（1）基础设施

在 Spark 中有很多基础设施，被 Spark 中的各种组件广泛使用。这些基础设施包括 Spark 配置（SparkConf）、Spark 内置的 RPC 框架（在早期 Spark 版本中 Spark 使用的是 Akka）、事件总线（ListenerBus）、度量系统。

① SparkConf 用于管理 Spark 应用程序的各种配置信息。

② Spark 内置的 RPC 框架使用 Netty 实现，有同步和异步的多种实现，Spark 各个组件间的通信都依赖于此 RPC 框架。

③ 如果说 RPC 框架是跨机器节点不同组件间的通信设施，那么事件总线就是 Spark Context 内部各个组件间使用事件——监听器模式异步调用的实现。

④ 度量系统由 Spark 中的多种度量源（Source）和多种度量输出（Sink）构成，完成对整个 Spark 集群中各个组件运行期状态的监控。

（2）SparkContext

通常而言，用户开发的 Spark 应用程序的提交与执行都离不开 SparkContext 的支持。在正式提交应用程序之前，首先需要初始化 SparkContext。SparkContext 隐藏了网络通信、分布式部署、消息通信、存储体系、计算引擎、度量系统、文件服务、Web UI 等内容，应用程序开发者只需要使用 SparkContext 提供的 API 完成功能开发。

（3）SparkEnv

Spark 执行环境 SparkEnv 是 Spark 中的 Task 运行所必需的组件。SparkEnv 内部封装了 RPC 环境（RpcEnv）、序列化管理器、广播管理器（BroadcastManager）、map 任务输出跟踪器（MapOutputTracker）、存储体系、度量系统（MetricsSystem）、输出提交协调器（OutputCommitCoordinator）等 Task 运行所需的各种组件。

（4）存储体系

Spark 优先考虑使用各节点的内存作为存储，当内存不足时才会考虑使用磁盘，这极大地减少了磁盘 I/O，提升了任务执行的效率，使得 Spark 适用于实时计算、迭代计算、流式计算等场景。在实际场景中，有些 Task 是存储密集型的，有些则是计算密集型的，所以有时候会造成存储空间很空闲，而计算空间的资源又很紧张。Spark 的内存存储空间与执

仅供非商业用途或交流学习使用

行存储空间之间的边界可以是“软”边界，因此资源紧张的一方可以借用另一方的空间，这既可以有效利用资源，又可以提高 Task 的执行效率。此外，Spark 的内存空间还提供了 Tungsten 的实现，直接操作操作系统的内存。由于 Tungsten 省去了在堆内分配 Java 对象，因此能更加有效地利用系统的内存资源，并且因为直接操作系统内存，空间的分配和释放也更迅速。在 Spark 早期版本还使用了以内存为中心的高容错的分布式文件系统 Alluxio（Tachyon）供用户进行选择。Alluxio 能够为 Spark 提供可靠的内存级的文件共享服务。

（5）调度系统

调度系统主要由 DAGScheduler 和 TaskScheduler 组成，它们都内置在 SparkContext 中。DAGScheduler 负责创建 Job、将 DAG 中的 RDD 划分到不同的 Stage、给 Stage 创建对应的 Task、批量提交 Task 等功能。TaskScheduler 负责按照 FIFO 或者 FAIR 等调度算法对批量 Task 进行调度；为 Task 分配资源；将 Task 发送到集群管理器的当前应用的 Executor 上，由 Executor 负责执行等工作。现如今，Spark 增加了 SparkSession 和 DataFrame 等新的 API，SparkSession 底层实际依然依赖于 SparkContext。

（6）计算引擎

计算引擎由内存管理器（MemoryManager）、Tungsten、任务内存管理器（TaskMemoryManager）、Task、外部排序器（ExternalSorter）、Shuffle 管理器（ShuffleManager）等组成。MemoryManager 除了对存储体系中的存储内存提供支持和管理外，还为计算引擎中的执行内存提供支持和管理。Tungsten 除用于存储外，也可以用于计算或执行。TaskMemoryManager 对分配给单个 Task 的内存资源进行更细粒度的管理和控制。ExternalSorter 用于在 map 端或 reduce 端对 ShuffleMapTask 计算得到的中间结果进行排序、聚合等操作。ShuffleManager 用于将各个分区对应的 ShuffleMapTask 产生的中间结果持久化到磁盘，并在 reduce 端按照分区远程拉取 ShuffleMapTask 产生的中间结果。



2. Spark 扩展功能

为了扩大应用范围，Spark 陆续增加了一些扩展功能，主要包括以下几项。

（1）Spark SQL

由于 SQL 具有普及率高、学习成本低等特点，为了扩大 Spark 的应用面，还增加了对 SQL 及 Hive 的支持。Spark SQL 的过程可以总结为：首先使用 SQL 语句解析器（SqlParser）将 SQL 转换为语法树（Tree），并且使用规则执行器（RuleExecutor）将一系列规则（Rule）应用到语法树，最终生成物理执行计划并执行的过程。其中，规则包括语法分析器（Analyzer）和优化器（Optimizer）。Hive 的执行过程与 SQL 类似。

（2）Spark Streaming

Spark Streaming 与 Apache Storm 类似，也用于流式计算。Spark Streaming 支持 Kafka、Flume、Kinesis 和简单的 TCP 套接字等多种数据输入源。输入流接收器（Receiver）负责接入数据，是接入数据流的接口规范。Dstream 是 Spark Streaming 中所有数据流的抽象，Dstream 可以被组织为 DStream Graph。Dstream 本质上由一系列连续的 RDD 组成。

仅供非商业用途或交流学习使用

（3）GraphX

Spark 提供的分布式图计算框架。GraphX 主要遵循整体同步并行计算模式（Bulk Synchronous Parallell，BSP）下的 Pregel 模型实现。GraphX 提供了对图 Graph 的抽象，Graph 由顶点（Vertex）、边（Edge）及继承了 Edge 的 EdgeTriplet（添加了 srcAttr 和 dstAttr，用来保存源顶点和目的顶点的属性）三种结构组成。GraphX 目前已经封装了最短路径、网页排名、连接组件、三角关系统计等算法的实现，用户可以选择使用。

（4）MLlib

Spark 提供的机器学习框架。机器学习是一门涉及概率论、统计学、逼近论、凸分析、算法复杂度理论等多领域的交叉学科。MLlib 目前已经提供了基础统计、分类、回归、决策树、随机森林、朴素贝叶斯、保序回归、协同过滤、聚类、维数缩减、特征提取与转型、频繁模式挖掘、预言模型标记语言、管道等多种数理统计、概率论、数据挖掘方面的数学算法。

小贴士：由于本书旨在着重分析 Spark Core 的原理，因此不会对 Spark 的扩展功能进行分析。对 Spark SQL、Spark Streaming、GraphX、MLlib 等内容感兴趣的读者，可以查阅其他文章或书籍，也可以选择阅读笔者所著的《深入理解 Spark：核心思想与源码分析》一书对上述内容的介绍。



2.3.2 Spark 模型设计

1. Spark 编程模型

正如 Hadoop 在介绍 MapReduce 编程模型时选择 word count 的例子，并且使用图形来说明一样，笔者也选择用图形展现 Spark 编程模型。

Spark 应用程序从编写到提交、执行、输出的整个过程如图 2-5 所示。

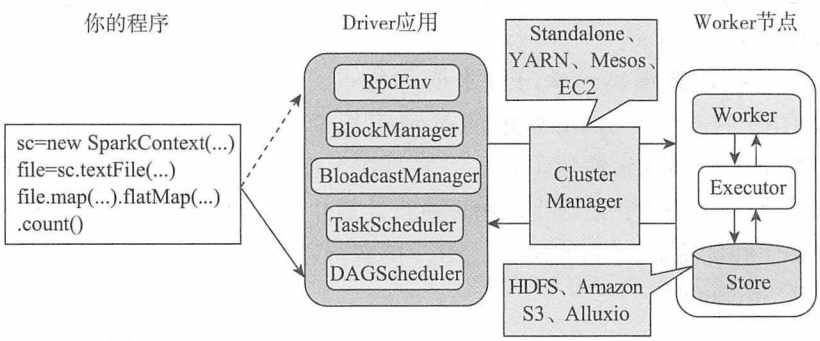


图 2-5　代码执行过程

图 2-5 中描述了 Spark 编程模型的关键环节，其步骤如下。

仅供非商业用途或交流学习使用

1）用户使用 SparkContext 提供的 API[⊖]编写 Driver 应用程序。此外，SparkSession、DataFrame、SQLContext、HiveContext 及 StreamingContext 都对 SparkContext 进行了封装，并提供了 DataFrame、SQL、Hive 及流式计算相关的 API。

2）使用 SparkContext 提交的用户应用程序，首先会通过 RpcEnv 向集群管理器（Cluster Manager）注册应用（Application）并且告知集群管理器需要的资源数量。集群管理器根据 Application 的需求，给 Application 分配 Executor 资源，并在 Worker 上启动 CoarseGrainedExecutorBackend 进程（该进程内部将创建 Executor）。Executor 所在的 CoarseGrainedExecutorBackend 进程在启动的过程中将通过 RpcEnv 直接向 Driver 注册 Executor 的资源信息，TaskScheduler 将保存已经分配给应用的 Executor 资源的地址、大小等相关信息。然后，SparkContext 根据各种转换 API，构建 RDD 之间的血缘关系和 DAG，RDD 构成的 DAG 将最终提交给 DAGScheduler。DAGScheduler 给提交的 DAG 创建 Job，并根据 RDD 的依赖性质将 DAG 划分为不同的 Stage。DAGScheduler 根据 Stage 内 RDD 的 Partition 数量创建多个 Task 并批量提交给 TaskScheduler。TaskScheduler 对批量的 Task 按照 FIFO 或 FAIR 调度算法进行调度，然后给 Task 分配 Executor 资源，最后将 Task 发送给 Executor 由 Executor 执行。此外，SparkContext 还会在 RDD 转换开始之前使用 BlockManager 和 BroadcastManager 将任务的 Hadoop 配置进行广播。

3）集群管理器（Cluster Manager）会根据应用的需求，给应用分配资源，即将具体任务分配到不同 Worker 节点上的多个 Executor 来处理任务的运行。Standalone、YARN、Mesos、EC2 等都可以作为 Spark 的集群管理器。

4）Task 在运行的过程中需要对一些数据（如中间结果、检查点等）进行持久化，Spark 支持选择 HDFS、Amazon S3、Alluxio（原名叫 Tachyon）等作为存储。

2. RDD 计算模型

RDD 可以看作是对各种数据计算模型的统一抽象，Spark 的计算过程主要是 RDD 的迭代计算过程，如图 2-6 所示。RDD 的迭代计算过程非常类似于管道。分区数量取决于 Partition 数量的设定，每个分区的数据只会在一个 Task 中计算。所有分区可以在多个机器节点的 Executor 上并行执行。

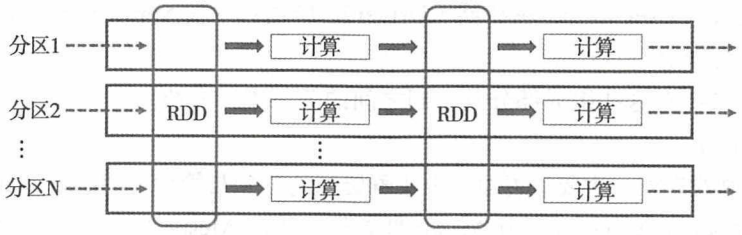


图 2-6　RDD 计算模型

⊖ 常用的有 textFile、sequenceFile、runJob、stop 等。

仅供非商业用途或交流学习使用

图 2-6 只是简单地从分区的角度将 RDD 的计算看作是管道，如果从 RDD 的血缘关系、Stage 划分的角度来看，由 RDD 构成的 DAG 经过 DAGScheduler 调度后，将变成如图 2-7 所示的样子。

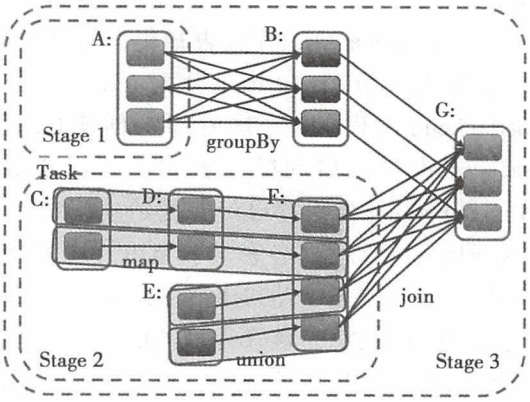


图 2-7 DAGScheduler 对由 RDD 构成的 DAG 进行调度

图 2-7 中共展示了 A、B、C、D、E、F、G 一共 7 个 RDD。每个 RDD 中的小方块代表一个分区，将会有一个 Task 处理此分区的数据。RDD A 经过 groupByKey 转换后得到 RDD B。RDD C 经过 map 转换后得到 RDD D。RDD D 和 RDD E 经过 union 转换后得到 RDD F。RDD B 和 RDD F 经过 join 转换后得到 RDD G。从图 2-7 中可以看到，map 和 union 生成的 RDD 与其上游 RDD 之间的依赖是 NarrowDependency，而 groupByKey 和 join 生成的 RDD 与其上游的 RDD 之间的依赖是 ShuffleDependency。由于 DAGScheduler 按照 ShuffleDependency 作为 Stage 的划分的依据，因此 A 被划入了 ShuffleMapStage 1；C、D、E、F 被划入了 ShuffleMapStage 2；B 和 G 被划入了 ResultStage 3。

2.4 Spark 基本架构

从集群部署的角度来看，Spark 集群由集群管理器（Cluster Manager）、工作节点（Worker）、执行器（Executor）、驱动器（Driver）、应用程序（Application）等部分组成，它们之间的整体关系如图 2-8 所示。

下面结合图 2-8 对这些组成部分及它们之间的关系进行介绍。

（1）Cluster Manager

Spark 的集群管理器，主要负责对整个集群资源的分配与管理。Cluster Manager 在 YARN 部署模式下为 ResourceManager；在 Mesos 部署模式下为 Mesos Master；在 Standalone 部署模式下为 Master。Cluster Manager 分配的资源属于一级分配，它将各个 Worker 上的内存、CPU 等资源分配给 Application，但是并不负责对 Executor 的资源分配。Standalone 部署模式

仅供非商业用途或交流学习使用

下的 Master 会直接给 Application 分配内存、CPU 及 Executor 等资源。目前，Standalone、YARN、Mesos、EC2 等都可以作为 Spark 的集群管理器。

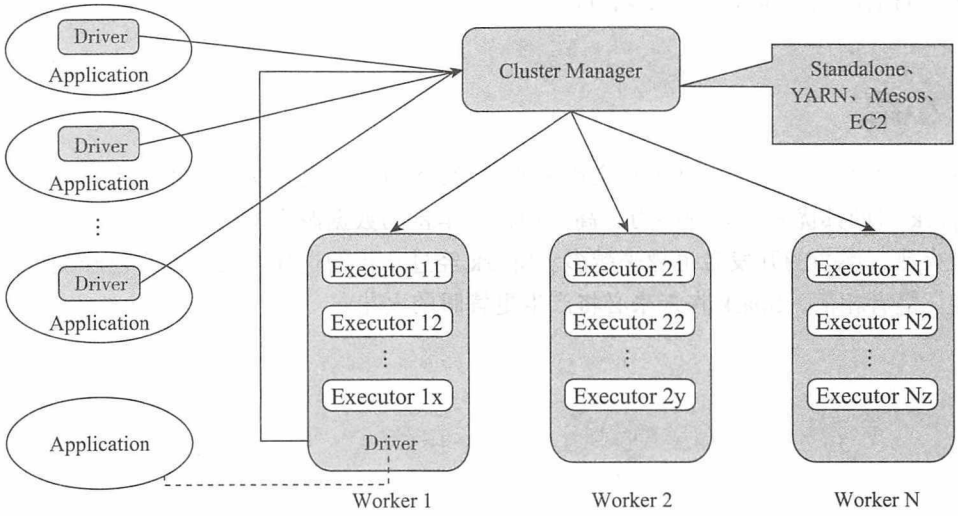


图 2-8　Spark 基本架构图

注意　这里提到了部署模式中的 Standalone、YARN、Mesos 等模式，读者暂时知道这些内容即可，第 9 章将对它们进行详细介绍。

（2）Worker

Spark 的工作节点。在 YARN 部署模式下实际由 NodeManager 替代。Worker 节点主要负责以下工作：将自己的内存、CPU 等资源通过注册机制告知 Cluster Manager；创建 Executor；将资源和任务进一步分配给 Executor；同步资源信息、Executor 状态信息给 Cluster Manager 等。在 Standalone 部署模式下，Master 将 Worker 上的内存、CPU 及 Executor 等资源分配给 Application 后，将命令 Worker 启动 CoarseGrainedExecutorBackend 进程（此进程会创建 Executor 实例）。

（3）Executor

执行计算任务的一线组件。主要负责任务的执行及与 Worker、Driver 的信息同步。

（4）Driver

Application 的驱动程序，Application 通过 Driver 与 Cluster Manager、Executor 进行通信。Driver 可以运行在 Application 中，也可以由 Application 提交给 Cluster Manager 并由 Cluster Manager 安排 Worker 运行。

（5）Application

用户使用 Spark 提供的 API 编写的应用程序，Application 通过 Spark API 将进行 RDD 的转换和 DAG 的构建，并通过 Driver 将 Application 注册到 Cluster Manager。Cluster

書籍是人类进步的阶梯

仅供非商业用途或交流学习使用

Manager 将会根据 Application 的资源需求，通过一级分配将 Executor、内存、CPU 等资源分配给 Application。Driver 通过二级分配将 Executor 等资源分配给每一个任务，Application 最后通过 Driver 告诉 Executor 运行任务。

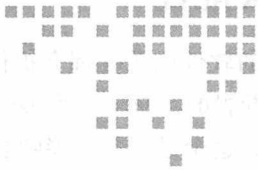
2.5 小结

每项技术的诞生都会由某种社会需求所驱动，Spark 正是在实时计算的大量需求下诞生的。Spark 借助其优秀的处理能力，高可用性，丰富的数据源支持等特点，在当前大数据领域变得火热，参与的开发者也越来越多。Spark 经过几年的迭代发展，如今已经提供了丰富的功能。笔者相信，Spark 在未来必将产生更耀眼的火花。



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用



第 3 章 Chapter 3

Spark 基础设施

“道生一，一生二，二生三，三生万物。”

——《道德经》



本章导读

我们生活的地球，是一个大千世界。地理上，既有寸草不生的撒哈拉大沙漠，又有河水泛滥的亚马逊森林；古诗中，既有王维所高歌的“大漠孤烟直，长河落日圆”，又有张继所低吟的“月落乌啼霜满天，江枫渔火对愁眠”；音乐中，既有古筝弹奏的典雅《高山流水》，又有笛子吹奏的民俗《小放牛》。道家认为道包含阴阳二气，阴或阳既有对立的一面，又有交汇的一面。阴阳交汇就产生了世间万物。用现代科学来讲，道就是整个宇宙的规律或法则，它包含一系列的自然法则，如阴阳、正负、矛盾等。这些法则将生成各种元素，元素经过化学的、物理的反应创造出天地万物。

同理，Spark 无论多么引人注目，究其根本，也脱离不了最基本、最简单的基础设施。Spark 的任何核心功能都离不开这些基础设施的构建，因此阅读本章内容将有助于我们理解 Spark 最底层的原理与实现。本书的核心篇以本章为始，也是秉着“深入浅出”的原则，让大家从基础知识慢慢进入核心内容的理解。

本章主要讲解的内容如下。

- Spark 配置。
- RPC 框架。
- 事件总线。
- 度量系统。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

3.1 Spark 配置

任何优秀的软件或服务都会提供一些配置参数，这些配置参数有些是内置的，有些则是可以由用户配置的。对于熟悉 Java 的开发人员来说，对 JVM 进行性能调优是一个经常需要面对的工作，这个过程常常伴随着各种 JVM 参数的调整与测试。之所以将这些参数交给具体的开发人员去调整，是因为软件或服务的提供者也无法保证给定的默认参数是最符合用户应用场景与软硬件环境的。举一个简单的例子：当用户的 QPS 发生变化时，对于 Web 服务的 JVM 来说也应当相应调整内存的大小或限制。

Spark 作为一款优秀的计算框架，也配备了各种各样的系统配置参数。SparkConf 是 Spark 的配置类，这个类在 Spark 的历史版本中已经存在很久了，Spark 中的每一个组件都直接或者间接地使用着它所存储的属性，这些属性都存储在如下的数据结构中。

```
private val settings = new ConcurrentHashMap[String, String]()
```

由以上代码的泛型[⊖]可以看出，Spark 的所有配置，无论是 key 还是 value 都是 String 类型。Spark 的配置通过以下 3 种方式获取。

- 来源于系统参数（即使用 System.getProperties 获取的属性）中以 spark. 作为前缀的那部分属性；
- 使用 SparkConf 的 API 进行设置；
- 从其他 SparkConf 中克隆。

下面将具体说明这 3 种方式的实现。



3.1.1 系统属性中的配置

在 SparkConf 中有一个 Boolean 类型的构造器属性 loadDefaults，当 loadDefaults 为 true 时，将会从系统属性中加载 Spark 配置，代码如下：

```
if (loadDefaults) {
  loadFromSystemProperties(false)
}

private[spark] def loadFromSystemProperties(silent: Boolean): SparkConf = {
  // 加载以spark.开头的系统属性
  for ((key, value) <- Utils.getSystemProperties if key.startsWith("spark.")) {
    set(key, value, silent)
  }
  this
}
```

以上代码调用了 Utils 工具类[⊜]的 getSystemProperties 方法，其作用为获取系统的键值

⊖ Scala 泛型的语法采用了方括号，而非 Java 中的尖括号。

⊜ Utils 是 Spark 中最常用的工具类，其每个方法实现的功能都比较单一，理解起来比较简单，所以本书只将相关的介绍单独放入附录 A 中。如果不去阅读 Utils 中各个方法的实现，对阅读本书主干内容也不会有太多影响。如果是 Spark 的初学者或者是刚刚接触 Scala 语言的开发者，还是建议阅读。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

对属性。loadFromSystemProperties 方法在获取了系统属性后，使用 Scala 守卫过滤出其中以“spark.”字符串为前缀的 key 和 value 并且调用 set 方法（见代码清单 3-1），最终设置到 settings 中。

代码清单3-1　SparkConf中set方法的实现

```
private[spark] def set(key: String, value: String, silent: Boolean): SparkConf = {
  if (key == null) {
    throw new NullPointerException("null key")
  }
  if (value == null) {
    throw new NullPointerException("null value for " + key)
  }
  if (!silent) {
    logDeprecationWarning(key)
  }
  settings.put(key, value)
  this
}
```

3.1.2　使用 SparkConf 配置的 API

给 SparkConf 添加配置的一种常见方式是使用 SparkConf 中提供的 API。其中有些 API 最终实际调用了 set 的重载方法，如代码清单 3-2 所示。



代码清单3-2　SparkConf中重载的set方法

```
def set(key: String, value: String): SparkConf = {
  set(key, value, false)
}
```

可以看到代码清单 3-2 中的 set 方法实际也是调用了代码清单 3-1 中的 set 方法。

SparkConf 中的 setMaster、setAppName、setJars、setExecutorEnv、setSparkHome、setAll 等方法最终都是通过代码清单 3-2 中的 set 方法完成 Spark 配置的，本书以其中最为常用的 setMaster 和 setAppName 为例，用代码清单 3-3 和代码清单 3-4 来展示它们的实现。

代码清单3-3　设置Spark的部署模式的配置方法setMaster

```
def setMaster(master: String): SparkConf = {
  set("spark.master", master)
}
```

代码清单3-4　设置Spark的应用名称的配置方法setAppName

```
def setAppName(name: String): SparkConf = {
  set("spark.app.name", name)
}
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

3.1.3 克隆 SparkConf 配置

有些情况下，同一个 SparkConf 实例中的配置信息需要被 Spark 中的多个组件共用，例如，组件 A 中存在一个 SparkConf 实例 a，组件 B 中也需要实例 a 中的配置信息，这时该如何处理？我们往往首先想到的方法是将 SparkConf 实例定义为全局变量或者通过参数传递给其他组件，但是这会引入并发问题。虽然 settings 是线程安全的 ConcurrentHashMap 类，而且 ConcurrentHashMap 也被证明是高并发下性能表现不错的数据结构，但是只要存在并发，就一定会有性能的损失问题。我们可以新建一个 SparkConf 实例 b，并将 a 中的配置信息全部拷贝到 b 中，这种方式显然不是最优雅的，复制代码会散落在程序的各个角落。现在是时候阅读下 SparkConf 的构造器了，代码如下所示：

```
class SparkConf(loadDefaults: Boolean) extends Cloneable with Logging with
    Serializable {
//省略无关代码
def this() = this(true)
```

SparkConf 继承了 Cloneable 特质并实现了 clone 方法，clone 方法（见代码清单 3-5）的实现跟我们所讨论的方式是一样的，并且通过 Cloneable 特质提高了代码的可复用性。

代码清单3-5 克隆SparkConf配置

```
override def clone: SparkConf = {
  val cloned = new SparkConf(false)
  settings.entrySet().asScala.foreach { e =>
    cloned.set(e.getKey(), e.getValue(), true)
  }
  cloned
}
```

这样我们就可以在任何想要使用 SparkConf 的地方使用克隆方式来优雅地编程了。

3.2 Spark 内置 RPC 框架

在 Spark 中很多地方都涉及网络通信，比如 Spark 各个组件间的消息互通、用户文件与 Jar 包的上传、节点间的 Shuffle 过程、Block 数据的复制与备份等。在 Spark 0.x.x 与 Spark 1.x.x 版本中，组件间的消息通信主要借助于 Akka[⊖]，使用 Akka 可以轻松地构建强有力的高并发与分布式应用。但是 Akka 在 Spark 2.0.0 版本中被移除了，Spark 官网文档对此的描述为：“Akka 的依赖被移除了，因此用户可以使用任何版本的 Akka 来编程了。”Spark 团队的决策者或许认为对 Akka 具体版本的依赖，限制了用户对 Akka 不同版本的使用。尽管如此，笔者依然认为 Akka 是一款非常优秀的开源分布式系统，我参与的一些 Java

⊖ Akka 是基于 Actor 并发编程模型实现的并发的分布式框架。Akka 是用 Scala 语言编写的，它提供了 Java 和 Scala 两种语言的 API，减少了开发人员对并发的细节处理，并保证分布式调用的最终一致性。在附录 B 中有关于 Akka 的进一步介绍，感兴趣的读者不妨一读。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

Application 或者 Java Web 就利用 Akka 的丰富特性实现了分布式一致性、最终一致性及分布式事务等分布式环境面对的问题。在 Spark 1.x.x 版本中，用户文件与 Jar 包的上传采用了由 Jetty[⊖]实现的 HttpFileServer，但在 Spark 2.0.0 版本中它也被废弃了，现在使用的是基于 Spark 内置 RPC 框架的 NettyStreamManager。节点间的 Shuffle 过程和 Block 数据的复制与备份这两个部分在 Spark 2.0.0 版本中依然沿用了 Netty[⊜]，通过对接口和程序进行重新设计，将各个组件间的消息互通、用户文件与 Jar 包的上传等内容统一纳入 Spark 的 RPC 框架体系中。

我们先来看看 RPC 框架的基本架构，如图 3-1 所示。

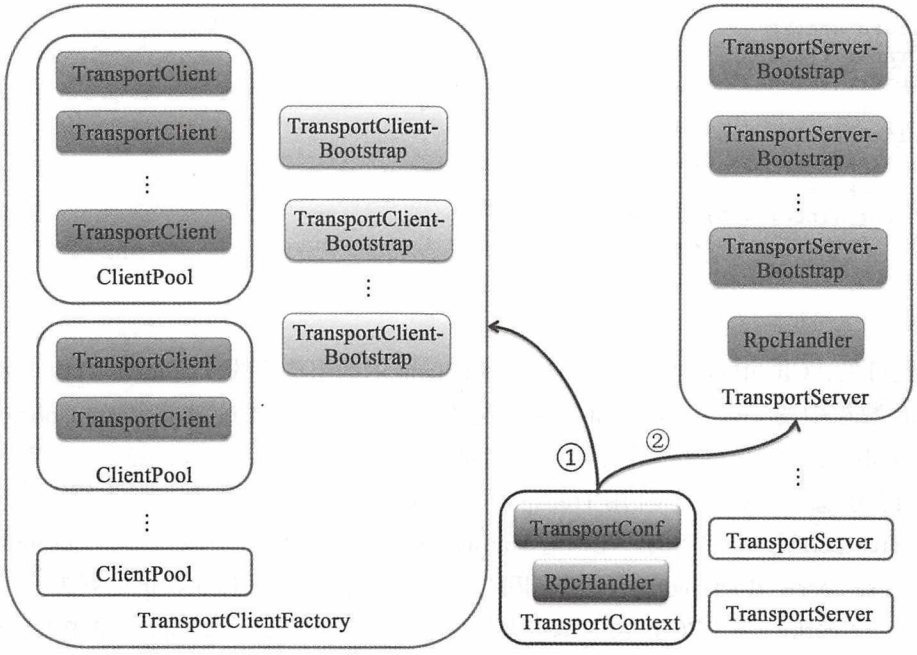


图 3-1　Spark 内置 RPC 框架的基本架构

TransportContext 内部包含传输上下文的配置信息 TransportConf 和对客户端请求消息进行处理的 RpcHandler。TransportConf 在创建 TransportClientFactory 和 TransportServer 时都是必需的，而 RpcHandler 只用于创建 TransportServer。TransportClientFactory 是 RPC 客

⊖ Jetty 是一个开源的 Servlet 容器，它为基于 Java 的 Web 容器，例如 JSP 和 Servlet 提供运行环境。Jetty 是使用 Java 语言编写的，它的 API 以一组 Jar 包的形式发布。开发人员可以将 Jetty 容器实例化成一个对象，可以迅速为一些独立运行的 Java 应用提供网络和 Web 连接。在附录 C 中有对 Jetty 的简单介绍，感兴趣的读者可以选择阅读。

⊜ Netty 是由 Jboss 提供的一个基于 NIO 的客户、服务端编程框架，使用 Netty 可以确保你快速、简单地开发出一个网络应用，例如实现了某种协议的客户、服务端应用。附录 G 中有对 Netty 的简单介绍，感兴趣的读者可以一读。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

户端的工厂类。TransportServer 是 RPC 服务端的实现。图中记号的含义如下。

- 记号①表示通过调用 TransportContext 的 createClientFactory 方法创建传输客户端工厂 TransportClientFactory 的实例。在构造 TransportClientFactory 的实例时，还会传递客户端引导程序 TransportClientBootstrap 的列表。此外，TransportClientFactory 内部还存在针对每个 Socket 地址的连接池 ClientPool，这个连接池缓存的定义如下：

```
private final ConcurrentHashMap<SocketAddress, ClientPool> connectionPool;
```

ClientPool 的类型定义如下：

```
private static class ClientPool {
  TransportClient[] clients;
  Object[] locks;

  ClientPool(int size) {
    clients = new TransportClient[size];
    locks = new Object[size];
    for (int i = 0; i < size; i++) {
      locks[i] = new Object();
    }
  }
}
```

由此可见，ClientPool 实际是由 TransportClient 的数组构成的，而 locks 数组中的 Object 与 clients 数组中的 TransportClient 按照数组索引一一对应，通过对每个 TransportClient 分别采用不同的锁，降低并发情况下线程间对锁的争用，进而减少阻塞，提高并发度。

- 记号②表示通过调用 TransportContext 的 createServer 方法创建传输服务端 TransportServer 的实例。在构造 TransportServer 的实例时，需要传递 TransportContext、host、port、RpcHandler 及服务端引导程序 TransportServerBootstrap 的列表。

有了对 Spark 内置 RPC 框架的基本架构的了解，现在正式介绍 Spark 的 RPC 框架所包含的各个组件。

- TransportContext：传输上下文，包含了用于创建传输服务端（TransportServer）和传输客户端工厂（TransportClientFactory）的上下文信息，并支持使用 TransportChannelHandler 设置 Netty 提供的 SocketChannel 的 Pipeline 的实现。
- TransportConf：传输上下文的配置信息。
- RpcHandler：对调用传输客户端（TransportClient）的 sendRPC 方法发送的消息进行处理的程序。
- MessageEncoder：在将消息放入管道前，先对消息内容进行编码，防止管道另一端读取时丢包和解析错误。
- MessageDecoder：对从管道中读取的 ByteBuf 进行解析，防止丢包和解析错误。
- TransportFrameDecoder：对从管道中读取的 ByteBuf 按照数据帧进行解析。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

- RpcResponseCallback：RpcHandler 对请求的消息处理完毕后进行回调的接口。
- TransportClientFactory：创建 TransportClient 的传输客户端工厂类。
- ClientPool：在两个对等节点间维护的关于 TransportClient 的池子。ClientPool 是 TransportClientFactory 的内部组件。
- TransportClient：RPC 框架的客户端，用于获取预先协商好的流中的连续块。TransportClient 旨在允许有效传输大量数据，这些数据将被拆分成几百 KB 到几 MB 的块。当 TransportClient 处理从流中获取的块时，实际的设置是在传输层之外完成的。sendRPC 方法能够在客户端和服务端的同一水平线的通信进行这些设置。
- TransportClientBootstrap：当服务端响应客户端连接时在客户端执行一次的引导程序。
- TransportRequestHandler：用于处理客户端的请求并在写完块数据后返回的处理程序。
- TransportResponseHandler：用于处理服务端的响应，并且对发出请求的客户端进行响应的处理程序。
- TransportChannelHandler：代理由 TransportRequestHandler 处理的请求和由 TransportResponseHandler 处理的响应，并加入传输层的处理。
- TransportServerBootstrap：当客户端连接到服务端时在服务端执行一次的引导程序。
- TransportServer：RPC 框架的服务端，提供高效的、低级别的流服务。



拓展知识：为什么需要 MessageEncoder 和 MessageDecoder？因为在基于流的传输里（比如 TCP/IP），接收到的数据首先会被存储到一个 socket 接收缓冲里。不幸的是，基于流的传输并不是一个数据包队列，而是一个字节队列。即使你发送了 2 个独立的数据包，操作系统也不会作为 2 个消息处理，而仅仅认为是一连串的字节，因此不能保证远程写入的数据会被准确地读取。举个例子，我们假设操作系统的 TCP/TP 协议栈已经接收了 3 个数据包：ABC、DEF、GHI。由于基于流传输的协议的这种统一的性质，你的应用程序在读取数据的时候有很大的可能性被分成下面的片段：AB、CDEFG、H、I。因此，接收方不管是客户端还是服务端，都应该把接收到的数据整理成一个或者多个更有意义并且让程序的逻辑更好理解的数据。

3.2.1　RPC 配置 TransportConf

上文提到 TransportContext 中的 TransportConf 给 Spark 的 RPC 框架提供配置信息，它有两个成员属性——配置提供者 conf 和配置的模块名称 module。这两个属性的定义如下：

```
private final ConfigProvider conf;
private final String module;
```

其中 conf 是真正的配置提供者，其类型 ConfigProvider 是一个抽象类，如代码清单 3-6 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

代码清单3-6 ConfigProvider的实现

```
public abstract class ConfigProvider {
public abstract String get(String name);

public String get(String name, String defaultValue) {
  try {
    return get(name);
  } catch (NoSuchElementException e) {
    return defaultValue;
  }
}

public int getInt(String name, int defaultValue) {
  return Integer.parseInt(get(name, Integer.toString(defaultValue)));
}

public long getLong(String name, long defaultValue) {
  return Long.parseLong(get(name, Long.toString(defaultValue)));
}

public double getDouble(String name, double defaultValue) {
  return Double.parseDouble(get(name, Double.toString(defaultValue)));
}

public boolean getBoolean(String name, boolean defaultValue) {
    return Boolean.parseBoolean(get(name, Boolean.toString(defaultValue)));
  }
}
```



从代码清单3-6可以看到，ConfigProvider中包括get、getInt、getLong、getDouble、getBoolean等方法，这些方法都是基于抽象方法get获取值，经过一次类型转换而实现。这个抽象的get方法将需要子类去实现。

Spark通常使用SparkTransportConf创建TransportConf，其实现如代码清单3-7所示。

代码清单3-7 SparkTransportConf的实现

```
object SparkTransportConf {
  private val MAX_DEFAULT_NETTY_THREADS = 8
  def fromSparkConf(_conf: SparkConf, module: String, numUsableCores: Int = 0):
     TransportConf = {
    val conf = _conf.clone
    val numThreads = defaultNumThreads(numUsableCores)
    conf.setIfMissing(s"spark.$module.io.serverThreads", numThreads.toString)
    conf.setIfMissing(s"spark.$module.io.clientThreads", numThreads.toString)

    new TransportConf(module, new ConfigProvider {
      override def get(name: String): String = conf.get(name)
    })
  }
  private def defaultNumThreads(numUsableCores: Int): Int = {
    val availableCores =
      if (numUsableCores > 0) numUsableCores else Runtime.getRuntime.
          availableProcessors()
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
      math.min(availableCores, MAX_DEFAULT_NETTY_THREADS)
    }
  }
```

从代码清单 3-7 看到，可以使用 SparkTransportConf 的 fromSparkConf 方法来构造 TransportConf。传递的三个参数分别为 SparkConf、模块名 module 及可用的内核数 numUsableCores。如果 numUsableCores 小于等于 0，那么线程数是系统可用处理器的数量，不过系统的内核数不可能全部用于网络传输，所以这里将分配给网络传输的内核数量最多限制在 8 个。最终确定的线程数将用于设置客户端传输线程数（spark.$module.io.clientThreads 属性）和服务端传输线程数（spark.$module.io.serverThreads 属性）。fromSparkConf 最终构造 TransportConf 对象时传递的 ConfigProvider 为实现了 get 方法的匿名的内部类，get 的实现实际是代理了 SparkConf 的 get 方法。

3.2.2　RPC 客户端工厂 TransportClientFactory

TransportClientFactory 是创建 TransportClient 的工厂类。在说明图 3-1 中的记号①时提到过，TransportContext 的 createClientFactory 方法可以创建 TransportClientFactory 的实例，其实现如代码清单 3-8 所示。

代码清单3-8　创建传输客户端工厂

```
public TransportClientFactory createClientFactory(List<TransportClientBootstrap>
bootstraps) {
  return new TransportClientFactory(this, bootstraps);
}

public TransportClientFactory createClientFactory() {
  return createClientFactory(Lists.<TransportClientBootstrap>newArrayList());
}
```

可以看到，TransportContext 中有两个重载的 createClientFactory 方法，它们最终在构造 TransportClientFactory 时都会传递两个参数：TransportContext 和 TransportClientBootstrap 列表。TransportClientFactory 构造器的实现如代码清单 3-9 所示。

代码清单3-9　TransportClientFactory的构造器

```
public TransportClientFactory(
    TransportContext context,
    List<TransportClientBootstrap> clientBootstraps) {
  this.context = Preconditions.checkNotNull(context);
  this.conf = context.getConf();
  this.clientBootstraps = Lists.newArrayList(Preconditions.checkNotNull(clientBo
      otstraps));
  this.connectionPool = new ConcurrentHashMap<>();
  this.numConnectionsPerPeer = conf.numConnectionsPerPeer();
  this.rand = new Random();

  IOMode ioMode = IOMode.valueOf(conf.ioMode());
```

仅供非商业用途或交流学习使用


```
    this.socketChannelClass = NettyUtils.getClientChannelClass(ioMode);
    this.workerGroup = NettyUtils.createEventLoop(
        ioMode,
        conf.clientThreads(),
        conf.getModuleName() + "-client");
    this.pooledAllocator = NettyUtils.createPooledByteBufAllocator(
      conf.preferDirectBufs(), false /* allowCache */, conf.clientThreads());
  }
```

TransportClientFactory 构造器中的各个变量如下。

- context：参数传递的 TransportContext 的引用。
- conf：指 TransportConf，这里通过调用 TransportContext 的 getConf 获取。
- clientBootstraps：参数传递的 TransportClientBootstrap 列表。
- connectionPool：针对每个 Socket 地址的连接池 ClientPool 的缓存。connectionPool 的数据结构较为复杂，为便于读者理解，这里以图 3-2 来表示 connectionPool 的数据结构。

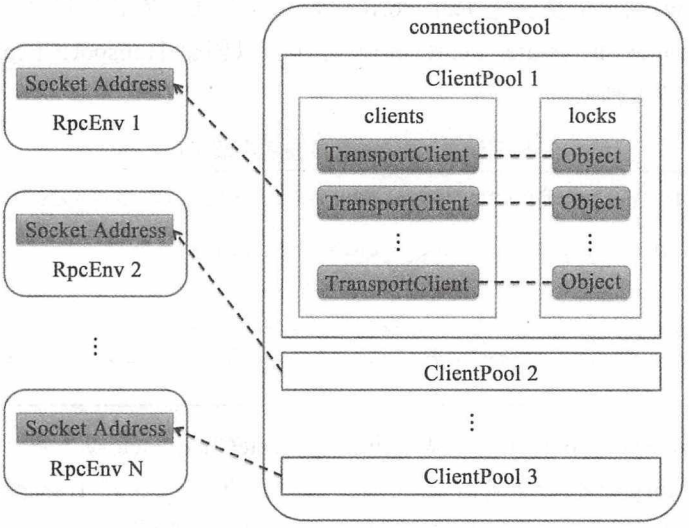


图 3-2 TransportClientFactory 的 connectionPool

- numConnectionsPerPeer：从 TransportConf 获取的 key 为“spark.+模块名+.io.numConnectionsPerPeer”的属性值。此属性值用于指定对等节点间的连接数。这里的模块名实际为 TransportConf 的 module 字段。Spark 的很多组件都利用 RPC 框架构建，它们之间按照模块名区分，例如，RPC 模块的 key 为“spark.rpc.io.numConnectionsPerPeer”。
- rand：对 Socket 地址对应的连接池 ClientPool 中缓存的 TransportClient 进行随机选择，对每个连接做负载均衡。
- ioMode：IO 模式，即从 TransportConf 获取 key 为“spark.+模块名+.io.mode”的

仅供非商业用途或交流学习使用

属性值。默认值为 NIO，Spark 还支持 EPOLL。

- socketChannelClass：客户端 Channel 被创建时使用的类，通过 ioMode 来匹配，默认为 NioSocketChannel，Spark 还支持 EpollEventLoopGroup。
- workerGroup：根据 Netty 的规范，客户端只有 worker 组，所以此处创建 workerGroup。workerGroup 的实际类型是 NioEventLoopGroup。
- pooledAllocator：汇集 ByteBuf 但对本地线程缓存禁用的分配器。

TransportClientFactory 里大量使用了 NettyUtils，关于 NettyUtils 的具体实现，请参看附录 G[⊖]。



NIO 是 Java 中 New IO 的简称，其特点包括为所有的原始类型提供（Buffer）缓冲支持；字符集编码解码解决方案；提供一个新的原始 I/O 抽象 Channel，支持锁和内存映射文件的文件访问接口；提供多路非阻塞式 (non-blocking) 的高伸缩性网络 I/O。其具体使用属于 Java 语言的范畴，本文不过多介绍。

1. 客户端引导程序 TransportClientBootstrap

TransportClientFactory 的 clientBootstraps 属性是 TransportClientBootstrap 的列表。TransportClientBootstrap 是在 TransportClient 上执行的客户端引导程序，主要对连接建立时进行一些初始化的准备（例如验证、加密）。TransportClientBootstrap 所做的操作往往是昂贵的，好在建立的连接可以重用。TransportClientBootstrap 的接口定义如代码清单 3-10 所示。



代码清单3-10　TransportClientBootstrap的定义

```
public interface TransportClientBootstrap {
void doBootstrap(TransportClient client, Channel channel) throws
RuntimeException;
}
```

TransportClientBootstrap 有两个实现类：EncryptionDisablerBootstrap 和 SaslClientBootstrap。为了对 TransportClientBootstrap 的作用能有更深的了解，这里以 EncryptionDisablerBootstrap 为例，EncryptionDisablerBootstrap 的实现如代码清单 3-11 所示。

代码清单3-11　EncryptionDisablerBootstrap的实现

```
private static class EncryptionDisablerBootstrap implements Transport-
    ClientBootstrap {
  @Override
  public void doBootstrap(TransportClient client, Channel channel) {
    channel.pipeline().remove(SaslEncryption.ENCRYPTION_HANDLER_NAME);
  }
}
```

⊖ Spark 将对 Netty 框架的使用细节都封装在 NettyUtils 工具类中，由于 Netty 的 API 使用不属于本书主要阐述的内容，故此放入附录 G 中，对 Netty 的使用感兴趣的读者可以选择阅读。

仅供非商业用途或交流学习使用

根据代码清单 3-11 可以看到，EncryptionDisablerBootstrap 的作用是移除客户端管道中的 SASL 加密。

2. 创建 RPC 客户端 TransportClient

有了 TransportClientFactory，Spark 的各个模块就可以使用它创建 RPC 客户端 TransportClient 了。每个 TransportClient 实例只能和一个远端的 RPC 服务通信，所以 Spark 中的组件如果想要和多个 RPC 服务通信，就需要持有多个 TransportClient 实例。创建 TransportClient 的方法如代码清单 3-12 所示（实际为从缓存中获取 TransportClient）。

代码清单3-12　从缓存中获取TransportClient

```
public TransportClient createClient(String remoteHost, int remotePort)
    throws IOException, InterruptedException {
  // 创建InetSocketAddress
  final InetSocketAddress unresolvedAddress =
    InetSocketAddress.createUnresolved(remoteHost, remotePort);

  ClientPool clientPool = connectionPool.get(unresolvedAddress);
  if (clientPool == null) {
    connectionPool.putIfAbsent(unresolvedAddress, new ClientPool(numConnectionsP
        erPeer));
    clientPool = connectionPool.get(unresolvedAddress);
  }

  // 随机选择一个TransportClient
  int clientIndex = rand.nextInt(numConnectionsPerPeer);
  TransportClient cachedClient = clientPool.clients[clientIndex];

  if (cachedClient != null && cachedClient.isActive()) {// 获取并返回激活的
                                                        //TransportClient
    TransportChannelHandler handler = cachedClient.getChannel().pipeline()
      .get(TransportChannelHandler.class);
    synchronized (handler) {
      handler.getResponseHandler().updateTimeOfLastRequest();
    }

    if (cachedClient.isActive()) {
      logger.trace("Returning cached connection to {}: {}",
        cachedClient.getSocketAddress(), cachedClient);
      return cachedClient;
    }
  }

  final long preResolveHost = System.nanoTime();
  final InetSocketAddress resolvedAddress = new InetSocketAddress(remoteHost,
      remotePort);
  final long hostResolveTimeMs = (System.nanoTime() - preResolveHost) / 1000000;
  if (hostResolveTimeMs > 2000) {
    logger.warn("DNS resolution for {} took {} ms", resolvedAddress, host
        ResolveTimeMs);
  } else {
    logger.trace("DNS resolution for {} took {} ms", resolvedAddress, host
```

仅供非商业用途或交流学习使用


```
            ResolveTimeMs);
        }
        // 创建并返回TransportClient对象
        synchronized (clientPool.locks[clientIndex]) {
          cachedClient = clientPool.clients[clientIndex];

          if (cachedClient != null) {
            if (cachedClient.isActive()) {
              logger.trace("Returning cached connection to {}: {}", resolvedAddress,
                  cachedClient);
              return cachedClient;
            } else {
              logger.info("Found inactive connection to {}, creating a new one.",
                  resolvedAddress);
            }
          }
          clientPool.clients[clientIndex] = createClient(resolvedAddress);
          return clientPool.clients[clientIndex];
        }
      }
```

从代码清单 3-12 得知，创建 TransportClient 的步骤如下。

1）调用 InetSocketAddress 的静态方法 createUnresolved 构建 InetSocketAddress（这种方式创建 InetSocketAddress，可以在缓存中已经有 TransportClient 时避免不必要的域名解析），然后从 connectionPool 中获取与此地址对应的 ClientPool，如果没有，则需要新建 ClientPool，并放入缓存 connectionPool 中。

2）根据 numConnectionsPerPeer 的大小（使用“spark.+ 模块名 +.io.numConnections-PerPeer”属性配置），从 ClientPool 中随机选择一个 TransportClient。

3）如果 ClientPool 的 clients 数组中在随机产生的索引位置不存在 TransportClient 或者 TransportClient 没有激活，则进入第 5 步，否则对此 TransportClient 进行第 4 步的检查。

4）更新 TransportClient 的 channel 中配置的 TransportChannelHandler 的最后一次使用时间，确保 channel 没有超时，然后检查 TransportClient 是否是激活状态，最后返回此 TransportClient 给调用方。

5）由于缓存中没有 TransportClient 可用，于是调用 InetSocketAddress 的构造器创建 InetSocketAddress 对象（直接使用 InetSocketAddress 的构造器创建 InetSocketAddress 会进行域名解析），在这一步骤多个线程可能会产生竞态条件（由于没有同步处理，所以多个线程极有可能同时执行到此处，都发现缓存中没有 TransportClient 可用，于是都使用 InetSocketAddress 的构造器创建 InetSocketAddress）。

6）第 5 步创建 InetSocketAddress 的过程中产生的竞态条件如果不妥善处理，会产生线程安全问题，所以到了 ClientPool 的 locks 数组发挥作用的时候了。按照随机产生的数组索引，locks 数组中的锁对象可以对 clients 数组中的 TransportClient 一对一进行同步。即便之前产生了竞态条件，但是在这一步只能有一个线程进入临界区。在临界区内，先进入的线程调用重载的 createClient 方法创建 TransportClient 对象并放入 ClientPool 的 clients 数组

仅供非商业用途或交流学习使用

中。当率先进入临界区的线程退出临界区后，其他线程才能进入，此时发现 ClientPool 的 clients 数组中已经存在了 TransportClient 对象，那么将不再创建 TransportClient，而是直接使用它。

代码清单 3-12 的整个执行过程实际解决了 TransportClient 缓存的使用及 createClient 方法的线程安全问题，并没有涉及创建 TransportClient 的实现。TransportClient 的创建过程在重载的 createClient 方法（见代码清单 3-13）中实现。

代码清单3-13 创建TransportClient

```
private TransportClient createClient(InetSocketAddress address)
    throws IOException, InterruptedException {
  logger.debug("Creating new connection to {}", address);
  // 构建根引导程序Bootstrap并对其进行配置
  Bootstrap bootstrap = new Bootstrap();
  bootstrap.group(workerGroup)
    .channel(socketChannelClass)
    .option(ChannelOption.TCP_NODELAY, true)
    .option(ChannelOption.SO_KEEPALIVE, true)
    .option(ChannelOption.CONNECT_TIMEOUT_MILLIS, conf.connectionTimeoutMs())
    .option(ChannelOption.ALLOCATOR, pooledAllocator);

  final AtomicReference<TransportClient> clientRef = new AtomicReference<>();
  final AtomicReference<Channel> channelRef = new AtomicReference<>();
  // 为根引导程序设置管道初始化回调函数
  bootstrap.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) {
      TransportChannelHandler clientHandler = context.initializePipeline(ch);
      clientRef.set(clientHandler.getClient());
      channelRef.set(ch);
    }
  });

  long preConnect = System.nanoTime();
  ChannelFuture cf = bootstrap.connect(address);// 使用根引导程序连接远程服务器
  if (!cf.await(conf.connectionTimeoutMs())) {
    throw new IOException(
      String.format("Connecting to %s timed out (%s ms)", address, conf.
          connectionTimeoutMs()));
  } else if (cf.cause() != null) {
    throw new IOException(String.format("Failed to connect to %s", address),
        cf.cause());
  }

  TransportClient client = clientRef.get();
  Channel channel = channelRef.get();
  assert client != null : "Channel future completed successfully with null
      client";

  // Execute any client bootstraps synchronously before marking the Client as
      successful.
  long preBootstrap = System.nanoTime();
```

仅供非商业用途或交流学习使用


```
    logger.debug("Connection to {} successful, running bootstraps...", address);
    try {
      for (TransportClientBootstrap clientBootstrap : clientBootstraps) {
        clientBootstrap.doBootstrap(client, channel);// 给TransportClient设置客户端引
            导程序
      }
    } catch (Exception e) { // catch non-RuntimeExceptions too as bootstrap may be
        written in Scala
      long bootstrapTimeMs = (System.nanoTime() - preBootstrap) / 1000000;
      logger.error("Exception while bootstrapping client after " + bootstrapTimeMs
          + " ms", e);
      client.close();
      throw Throwables.propagate(e);
    }
    long postBootstrap = System.nanoTime();

    logger.info("Successfully created connection to {} after {} ms ({} ms spent in
        bootstraps)",
      address, (postBootstrap - preConnect) / 1000000, (postBootstrap -
          preBootstrap) / 1000000);

    return client;
  }
```

从代码清单 3-13 得知，真正创建 TransportClient 的步骤如下。

1）构建根引导程序 Bootstrap 并对其进行配置。

2）为根引导程序设置管道初始化回调函数，此回调函数将调用 TransportContext 的 initializePipeline 方法初始化 Channel 的 pipeline。

3）使用根引导程序连接远程服务器，当连接成功对管道初始化时会回调初始化回调函数，将 TransportClient 和 Channel 对象分别设置到原子引用 clientRef 与 channelRef 中。

4）给 TransportClient 设置客户端引导程序，即设置 TransportClientFactory 中的 TransportClientBootstrap 列表。

5）返回此 TransportClient 对象。

3.2.3　RPC 服务端 TransportServer

TransportServer 是 RPC 框架的服务端，可提供高效、低级别的流服务。在说明图 3-1 中的记号②时提到过，TransportContext 的 createServer 方法用于创建 TransportServer，其实现如代码清单 3-14 所示。

代码清单3-14　创建RPC服务端

```
  public TransportServer createServer(int port, List<TransportServerBootstrap>
      bootstraps) {
    return new TransportServer(this, null, port, rpcHandler, bootstraps);
  }

  public TransportServer createServer(
```

仅供非商业用途或交流学习使用


```
    String host, int port, List<TransportServerBootstrap> bootstraps) {
  return new TransportServer(this, host, port, rpcHandler, bootstraps);
}

public TransportServer createServer(List<TransportServerBootstrap> bootstraps) {
  return createServer(0, bootstraps);
}

public TransportServer createServer() {
  return createServer(0, Lists.<TransportServerBootstrap>newArrayList());
}
```

代码清单 3-14 中列出了 4 个名为 createServer 的重载方法，但是它们最终调用了 TransportServer 的构造器（见代码清单 3-15）来创建 TransportServer 实例。

代码清单3-15　TransportServer的构造器

```
public TransportServer(
    TransportContext context,
    String hostToBind,
    int portToBind,
    RpcHandler appRpcHandler,
    List<TransportServerBootstrap> bootstraps) {
  this.context = context;
  this.conf = context.getConf();
  this.appRpcHandler = appRpcHandler;
  this.bootstraps = Lists.newArrayList(Preconditions.checkNotNull(bootstraps));

  try {
    init(hostToBind, portToBind);
  } catch (RuntimeException e) {
    JavaUtils.closeQuietly(this);
    throw e;
  }
}
```



TransportServer 的构造器中的各个变量如下。

- context：参数传递的 TransportContext 的引用。
- conf：指 TransportConf，这里通过调用 TransportContext 的 getConf 获取。
- appRpcHandler：RPC 请求处理器 RpcHandler。
- bootstraps：参数传递的 TransportServerBootstrap 列表。

TransportServer 的构造器（见代码清单 3-15）中调用了 init 方法，init 方法用于对 TransportServer 进行初始化，如代码清单 3-16 所示。

代码清单3-16　TransportServer的初始化

```
private void init(String hostToBind, int portToBind) {
  // 根据Netty的API文档，Netty服务端需同时创建bossGroup和workerGroup
  IOMode ioMode = IOMode.valueOf(conf.ioMode());
  EventLoopGroup bossGroup =
    NettyUtils.createEventLoop(ioMode, conf.serverThreads(), conf.getModuleName()
```

仅供非商业用途或交流学习使用


```
        + "-server");
    EventLoopGroup workerGroup = bossGroup;
    // 创建一个汇集ByteBuf但对本地线程缓存禁用的分配器
    PooledByteBufAllocator allocator = NettyUtils.createPooledByteBufAllocator(
      conf.preferDirectBufs(), true /* allowCache */, conf.serverThreads());
    // 创建Netty的服务端根引导程序并对其进行配置
    bootstrap = new ServerBootstrap()
      .group(bossGroup, workerGroup)
      .channel(NettyUtils.getServerChannelClass(ioMode))
      .option(ChannelOption.ALLOCATOR, allocator)
      .childOption(ChannelOption.ALLOCATOR, allocator);

    if (conf.backLog() > 0) {
      bootstrap.option(ChannelOption.SO_BACKLOG, conf.backLog());
    }
    if (conf.receiveBuf() > 0) {
      bootstrap.childOption(ChannelOption.SO_RCVBUF, conf.receiveBuf());
    }
    if (conf.sendBuf() > 0) {
      bootstrap.childOption(ChannelOption.SO_SNDBUF, conf.sendBuf());
    }
    // 为根引导程序设置管道初始化回调函数
    bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {
      @Override
      protected void initChannel(SocketChannel ch) throws Exception {
        RpcHandler rpcHandler = appRpcHandler;
        for (TransportServerBootstrap bootstrap : bootstraps) {
          rpcHandler = bootstrap.doBootstrap(ch, rpcHandler);
        }
        context.initializePipeline(ch, rpcHandler);
      }
    });
    // 给根引导程序绑定Socket的监听端口
    InetSocketAddress address = hostToBind == null ?
        new InetSocketAddress(portToBind): new InetSocketAddress(hostToBind,
          portToBind);
    channelFuture = bootstrap.bind(address);
    channelFuture.syncUninterruptibly();

    port = ((InetSocketAddress) channelFuture.channel().localAddress()).getPort();
    logger.debug("Shuffle server started on port: {}", port);
  }
```

代码清单 3-16 中 TransportServer 初始化的步骤如下。

1）创建 bossGroup 和 workerGroup。

小贴士：根据 Netty 的 API 文档，Netty 服务端需同时创建 bossGroup 和 workerGroup。

2）创建一个汇集 ByteBuf 但对本地线程缓存禁用的分配器。

3）调用 Netty 的 API 创建 Netty 的服务端根引导程序并对其进行配置。

4）为根引导程序设置管道初始化回调函数，此回调函数首先设置 TransportServer-

仅供非商业用途或交流学习使用

Bootstrap 到根引导程序中，然后调用 TransportContext 的 initializePipeline 方法初始化 Channel 的 pipeline。

5）给根引导程序绑定 Socket 的监听端口，最后返回监听的端口。

提示 代码清单 3-16 中使用了 NettyUtils 工具类的很多方法，在附录 G 中有对它们的详细介绍。EventLoopGroup、PooledByteBufAllocator、ServerBootstrap 都是 Netty 提供的 API，对于它们的更多介绍，请访问 http://netty.io/。

3.2.4 管道初始化

在代码清单 3-13 创建 TransportClient 和代码清单 3-16 对 TransportServer 初始化的实现中，都在管道初始化回调函数中调用了 TransportContext 的 initializePipeline 方法，initializePipeline 方法（见代码清单 3-17）将调用 Netty 的 API 对管道初始化。

代码清单3-17 管道初始化

```
public TransportChannelHandler initializePipeline(
    SocketChannel channel,
    RpcHandler channelRpcHandler) {
  try {
    TransportChannelHandler channelHandler = createChannelHandler(channel,
        channelRpcHandler);
    channel.pipeline()
      .addLast("encoder", ENCODER)
      .addLast(TransportFrameDecoder.HANDLER_NAME, NettyUtils.createFrame
          Decoder())
      .addLast("decoder", DECODER)
      .addLast("idleStateHandler", new IdleStateHandler(0, 0, conf.connection
          TimeoutMs() / 1000))
      .addLast("handler", channelHandler);
    return channelHandler;
  } catch (RuntimeException e) {
    logger.error("Error while initializing Netty pipeline", e);
    throw e;
  }
}
```



根据代码清单 3-17，initializePipeline 方法的执行步骤如下。

1）调用 createChannelHandler 方法创建 TransportChannelHandler，从 createChannelHandler 的实现（见代码清单 3-18）中可以看到，真正创建 TransportClient 是在这里发生的。通过 TransportClient 的构造过程看到 RpcHandler 与 TransportClient 毫无关系，TransportClient 只使用了 TransportResponseHandler。TransportChannelHandler 在服务端将代理 TransportRequestHandler 对请求消息进行处理，并在客户端代理 TransportResponseHandler 对响应消息进行处理。

仅供非商业用途或交流学习使用

代码清单3-18　创建TransportChannelHandler

```
private TransportChannelHandler createChannelHandler(Channel channel, RpcHandler
rpcHandler) {
  TransportResponseHandler responseHandler = new TransportResponseHandler(channel);
  TransportClient client = new TransportClient(channel, responseHandler);
   TransportRequestHandler requestHandler = new TransportRequestHandler(channel,
client,
    rpcHandler);
  return new TransportChannelHandler(client, responseHandler, requestHandler,
    conf.connectionTimeoutMs(), closeIdleConnections);
}
```

2）对管道进行设置，这里的 ENCODER（即 MessageEncoder）派生自 Netty 的 ChannelOutboundHandler 接　口；DECODER（即 MessageDecoder）、TransportChannelHandler 及 TransportFrameDecoder（由工具类 NettyUtils 的静态方法 createFrameDecoder 创建）派生自 Netty 的 ChannelInboundHandler 接口；IdleStateHandler 同时实现了 ChannelOutbound-Handler 和 ChannelInboundHandler 接口。根据 Netty 的 API 行为，通过 addLast 方法注册多个 Handler 时，ChannelInboundHandler 按照注册的先后顺序执行，ChannelOutboundHandler 按照注册的先后顺序逆序执行，因此在管道两端（无论是服务端还是客户端）处理请求和响应的流程如图 3-3 所示。

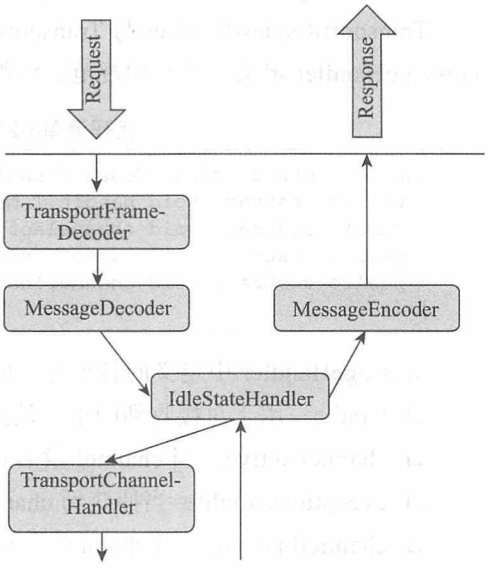


图 3-3　管道处理请求和响应的流程图

3.2.5　TransportChannelHandler 详解

TransportChannelHandler 实现了 Netty 的 ChannelInboundHandler[⊖]，以便对 Netty 管道中的消息进行处理。图 3-3 中的这些 Handler（除了 MessageEncoder）由于都实现了 ChannelInbound-Handler 接口，作为自定义的 ChannelInbound-Handler，所以都要重写 channelRead 方法。Netty 框架使用工作链模式来对每个 ChannelInboundHandler 的实现类的 channelRead 方法进行链式调用。TransportChannelHandler 实现的 channelRead 方法如代码清单 3-19 所示。

代码清单3-19　TransportChannelHandler的channelRead实现

```
@Override
public void channelRead(ChannelHandlerContext ctx, Object request) throws
```

⊖ ChannelInboundHandler 接口的实现及原理不属于本书要分析的内容，感兴趣的同学可以阅读 Netty 的官方文档或者研究 Netty 的源码。

仅供非商业用途或交流学习使用


```
Exception {
  if (request instanceof RequestMessage) {
    requestHandler.handle((RequestMessage) request);
  } else if (request instanceof ResponseMessage) {
    responseHandler.handle((ResponseMessage) request);
  } else {
    ctx.fireChannelRead(request);
  }
}
```

从代码清单 3-19 看到，当 TransportChannelHandler 读取的 request 是 RequestMessage 时，则将此消息的处理进一步交给 TransportRequestHandler，当读取的 request 是 ResponseMessage 时，则将此消息的处理进一步交给 TransportResponseHandler。

1. MessageHandler 的继承体系

TransportRequestHandler 与 TransportResponseHandler 都继承自抽象类 MessageHandler，MessageHandler 定义了子类的规范，详细定义如代码清单 3-20 所示。

代码清单3-20 MessageHandler规范

```
public abstract class MessageHandler<T extends Message> {
  public abstract void handle(T message) throws Exception;
  public abstract void channelActive();
  public abstract void exceptionCaught(Throwable cause);
  public abstract void channelInactive();
}
```



MessageHandler 中定义的各个方法的作用分别如下。

- handle：用于对接收到的单个消息进行处理。
- channelActive：当 channel 激活时调用。
- exceptionCaught：当捕获到 channel 发生异常时调用。
- channelInactive：当 channel 非激活时调用。

Spark 中 MessageHandler 类的继承体系如图 3-4 所示。

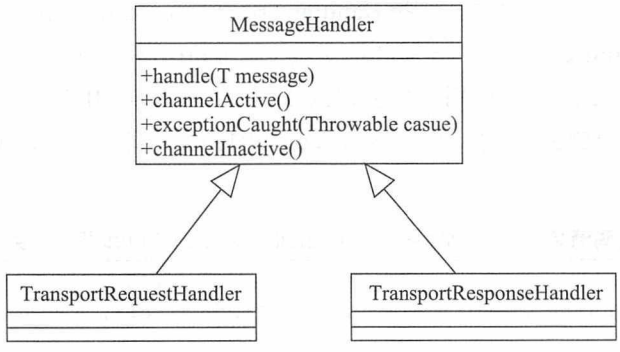


图 3-4 MessageHandler 类的继承体系

仅供非商业用途或交流学习使用

2. Message 的继承体系

根据代码清单 3-20 我们知道，MessageHandler 同时也是一个 Java 泛型类，其子类能处理的消息都派生自接口 Message。Message 的定义如代码清单 3-21 所示。

代码清单3-21　Message的定义

```
public interface Message extends Encodable {
  Type type();
  ManagedBuffer body();
  boolean isBodyInFrame();
```

Message 中定义的三个接口方法的作用分别如下。

- type：返回消息的类型。
- body：返回消息中可选的内容体。
- isBodyInFrame：用于判断消息的主体是否包含在消息的同一帧中。

Message 接口继承了 Encodable 接口，Encodable 的定义如代码清单 3-22 所示。

代码清单3-22　Encodable的定义

```
public interface Encodable {
  int encodedLength();
  void encode(ByteBuf buf);
}
```



实现 Encodable 接口的类将可以转换到一个 ByteBuf 中，多个对象将被存储到预先分配的单个 ByteBuf，所以这里的 encodedLength 用于返回转换的对象数量。下面一起来看看 Message 的类继承体系，如图 3-5 所示。

从图 3-5 看到，最终的消息实现类都直接或间接地实现了 RequestMessage 或 ResponseMessage 接口，其中 RequestMessage 的具体实现有 4 种，分别如下。

- ChunkFetchRequest：请求获取流的单个块的序列。
- RpcRequest：此消息类型由远程的 RPC 服务端进行处理，是一种需要服务端向客户端回复的 RPC 请求信息类型。
- OneWayMessage：此消息也需要由远程的 RPC 服务端进行处理，与 RpcRequest 不同的是，不需要服务端向客户端回复。
- StreamRequest：此消息表示向远程的服务发起请求，以获取流式数据。

由于 OneWayMessage 不需要响应，所以 ResponseMessage 对于成功或失败状态的实现各有 3 种，分别如下。

- ChunkFetchSuccess：处理 ChunkFetchRequest 成功后返回的消息。
- ChunkFetchFailure：处理 ChunkFetchRequest 失败后返回的消息。
- RpcResponse：处理 RpcRequest 成功后返回的消息。
- RpcFailure：处理 RpcRequest 失败后返回的消息。

仅供非商业用途或交流学习使用

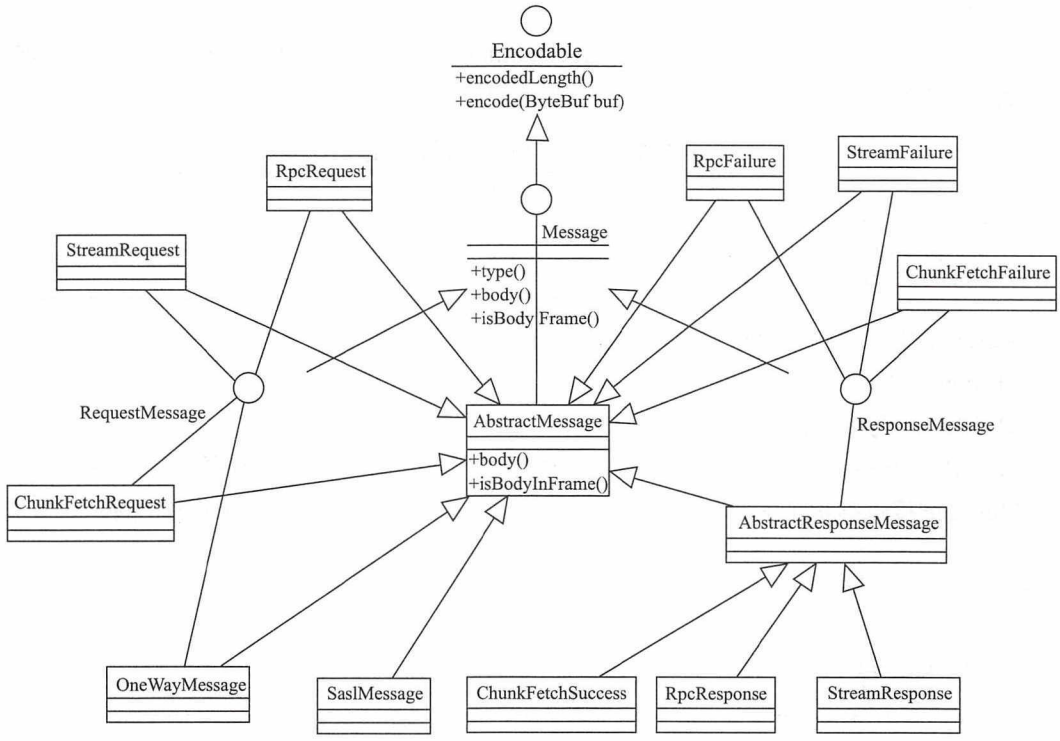


图 3-5 Message 的类继承体系

- StreamResponse：处理 StreamRequest 成功后返回的消息。
- StreamFailure：处理 StreamRequest 失败后返回的消息。

3. ManagedBuffer 的继承体系

回头再看看代码清单 3-21 中对 body 接口的定义，可以看到，其返回内容体的类型为 ManagedBuffer。ManagedBuffer 提供了由字节构成数据的不可变视图（也就是说，ManagedBuffer 并不存储数据，也不是数据的实际来源，这同关系型数据库的视图类似）。我们先来看看抽象类 ManagedBuffer 中对行为的定义，如代码清单 3-23 所示。

代码清单3-23 ManagedBuffer的定义

```
public abstract class ManagedBuffer {
  public abstract long size();
  public abstract ByteBuffer nioByteBuffer() throws IOException;
  public abstract InputStream createInputStream() throws IOException;
  public abstract ManagedBuffer retain();
  public abstract ManagedBuffer release();
  public abstract Object convertToNetty() throws IOException;
}
```

ManagedBuffer 中定义了 6 个方法，分别如下。

仅供非商业用途或交流学习使用

- size：返回数据的字节数。
- nioByteBuffer：将数据按照 NIO 的 ByteBuffer 类型返回。
- createInputStream：将数据按照 InputStream 返回。
- retain：当有新的使用者使用此视图时，增加引用此视图的引用数。
- release：当有使用者不再使用此视图时，减少引用此视图的引用数。当引用数为 0 时释放缓冲区。
- convertToNetty：将缓冲区的数据转换为 Netty 的对象，用来将数据写到外部。此方法返回的数据类型要么是 io.netty.buffer.ByteBuf，要么是 io.netty.channel.FileRegion。

ManagedBuffer 的具体实现有很多，我们可以通过图 3-6 来了解。

图 3-6 中列出了 ManagedBuffer 的 5 个实现类，其中 TestManagedBuffer 和 RecordingManagedBuffer 用于测试。NettyManagedBuffer 中的缓冲为 io.netty.buffer.ByteBuf，NioManagedBuffer 中的缓冲为 java.nio.ByteBuffer。NettyManagedBuffer 和 NioManagedBuffer 的实现都非常简单，留给读者自行阅读。本节挑选 FileSegmentManagedBuffer 作为 ManagedBuffer 具体实现的例子进行介绍。

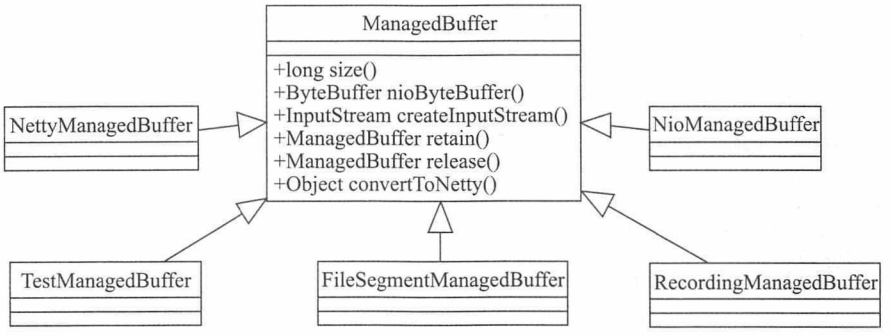


图 3-6　ManagedBuffer 的继承体系

FileSegmentManagedBuffer 的作用为获取一个文件中的一段，它一共有 4 个由 final 修饰的属性，全部都通过 FileSegmentManagedBuffer 的构造器传入属性值，这 4 个属性分别如下。

- conf：也就是 TransportConf。
- file：所要读取的文件。
- offset：所要读取文件的偏移量。
- length：所要读取的长度。

下面将逐个介绍 FileSegmentManagedBuffer 对于 ManagedBuffer 的实现。

- NIO 方式读取文件。FileSegmentManagedBuffer 实现的 nioByteBuffer 方法如代码清单 3-24 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

代码清单3-24 nioByteBuffer方法的实现

```
@Override
public ByteBuffer nioByteBuffer() throws IOException {
  FileChannel channel = null;
  try {
    channel = new RandomAccessFile(file, "r").getChannel();
    if (length < conf.memoryMapBytes()) {
      ByteBuffer buf = ByteBuffer.allocate((int) length);
      channel.position(offset);
      while (buf.remaining() != 0) {
        if (channel.read(buf) == -1) {
          throw new IOException(String.format("Reached EOF before filling buffer\n" +
            "offset=%s\nfile=%s\nbuf.remaining=%s",
            offset, file.getAbsoluteFile(), buf.remaining()));
        }
      }
      buf.flip();
      return buf;
    } else {
      return channel.map(FileChannel.MapMode.READ_ONLY, offset, length);
    }
  } catch (IOException e) {
    try {
      if (channel != null) {
        long size = channel.size();
        throw new IOException("Error in reading " + this + " (actual file length
            " + size + ")",
          e);
      }
    } catch (IOException ignored) {
      // ignore
    }
    throw new IOException("Error in opening " + this, e);
  } finally {
    JavaUtils.closeQuietly(channel);
  }
}
```



nioByteBuffer 的实现还是很简单的，主要利用 RandomAccessFile 获取 FileChannel，然后使用 java.nio.ByteBuffer 和 FileChannel 的 API 将数据写入缓冲区 java.nio.ByteBuffer 中。

- 文件流方式读取文件。FileSegmentManagedBuffer 实现的 createInputStream 方法如代码清单 3-25 所示。

代码清单3-25 createInputStream的实现

```
@Override
public InputStream createInputStream() throws IOException {
  FileInputStream is = null;
  try {
    is = new FileInputStream(file);
    ByteStreams.skipFully(is, offset);
    return new LimitedInputStream(is, length);
  } catch (IOException e) {
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    try {
      if (is != null) {
        long size = file.length();
        throw new IOException("Error in reading " + this + " (actual file length
            " + size + ")",
            e);
      }
    } catch (IOException ignored) {
      // ignore
    } finally {
      JavaUtils.closeQuietly(is);
    }
    throw new IOException("Error in opening " + this, e);
  } catch (RuntimeException e) {
    JavaUtils.closeQuietly(is);
    throw e;
  }
}
```

createInputStream 的实现还是很简单的，这里不过多介绍。

- 将数据转换为 Netty 对象。FileSegmentManagedBuffer 实现的 convertToNetty 方法如代码清单 3-26 所示。

代码清单3-26　convertToNetty的实现

```
@Override
public Object convertToNetty() throws IOException {
  if (conf.lazyFileDescriptor()) {
    return new DefaultFileRegion(file, offset, length);
  } else {
    FileChannel fileChannel = new FileInputStream(file).getChannel();
    return new DefaultFileRegion(fileChannel, offset, length);
  }
}
```

- 其他方法的实现。其他方法由于实现非常简单，所以这里就不一一列出了，感兴趣的读者可以自行查阅。

3.2.6　服务端 RpcHandler 详解

由于 TransportRequestHandler 实际是把请求消息交给 RpcHandler 进行进一步处理，所以这里对 RpcHandler 首先做个介绍。RpcHandler 是一个抽象类，定义了一些 RPC 处理器的规范，其主要实现如代码清单 3-27 所示。

代码清单3-27　RpcHandler的实现

```
public abstract class RpcHandler {
private static final RpcResponseCallback ONE_WAY_CALLBACK = new OneWayRpc-
    Callback();
public abstract void receive(
    TransportClient client,
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    ByteBuffer message,
    RpcResponseCallback callback);
public abstract StreamManager getStreamManager();
public void receive(TransportClient client, ByteBuffer message) {
  receive(client, message, ONE_WAY_CALLBACK);
}
public void channelActive(TransportClient client) { }
public void channelInactive(TransportClient client) { }
public void exceptionCaught(Throwable cause, TransportClient client) { }
private static class OneWayRpcCallback implements RpcResponseCallback {
  private static final Logger logger = LoggerFactory.getLogger(OneWayRpcCallback.
      class);
  @Override
  public void onSuccess(ByteBuffer response) {
    logger.warn("Response provided for one-way RPC.");
  }
  @Override
  public void onFailure(Throwable e) {
    logger.error("Error response provided for one-way RPC.", e);
  }
  }
}
```

代码清单 3-27 中 RpcHandler 的各个方法的作用如下。

- receive：这是一个抽象方法，用来接收单一的 RPC 消息，具体处理逻辑需要子类去实现。receive 接收 3 个参数，分别是 TransportClient、ByteBuffer 和 RpcResponseCallback。RpcResponseCallback 用于对请求处理结束后进行回调，无论处理结果是成功还是失败，RpcResponseCallback 都会被调用一次。RpcResponseCallback 的接口定义如下。

```
public interface RpcResponseCallback {
  void onSuccess(ByteBuffer response);
  void onFailure(Throwable e);
}
```

- 重载的 receive：只接收 TransportClient 和 ByteBuffer 两个参数，RpcResponseCallback 为默认的 ONE_WAY_CALLBACK，其类型为 OneWayRpcCallback，从代码清单 3-27 中 OneWayRpcCallback 的实现可以看出，其 onSuccess 和 onFailure 只是打印日志，并没有针对客户端做回复处理。
- channelActive：当与给定客户端相关联的 channel 处于活动状态时调用。
- channelInactive：当与给定客户端相关联的 channel 处于非活动状态时调用。
- exceptionCaught：当 channel 产生异常时调用。
- getStreamManager：获取 StreamManager，StreamManager 可以从流中获取单个的块，因此它也包含着当前正在被 TransportClient 获取的流的状态。

介绍完 RpcHandler，现在回到 TransportRequestHandler 的处理过程。TransportRequestHandler 处理以上 4 种 RequestMessage 的实现如代码清单 3-28 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

代码清单3-28　TransportRequestHandler的handle方法

```
@Override
public void handle(RequestMessage request) {
  if (request instanceof ChunkFetchRequest) {
    processFetchRequest((ChunkFetchRequest) request);
  } else if (request instanceof RpcRequest) {
    processRpcRequest((RpcRequest) request);
  } else if (request instanceof OneWayMessage) {
    processOneWayMessage((OneWayMessage) request);
  } else if (request instanceof StreamRequest) {
    processStreamRequest((StreamRequest) request);
  } else {
    throw new IllegalArgumentException("Unknown request type: " + request);
  }
}
```

结合代码清单 3-28，下面详细分析这 4 种类型请求的处理过程。

1. 处理块获取请求

processFetchRequest 方法用于处理 ChunkFetchRequest 类型的消息，其实现如代码清单 3-29 所示。

代码清单3-29　processFetchRequest的实现

```
private void processFetchRequest(final ChunkFetchRequest req) {
  if (logger.isTraceEnabled()) {
    logger.trace("Received req from {} to fetch block {}", getRemoteAddress
        (channel),
      req.streamChunkId);
  }
  ManagedBuffer buf;
  try {
    streamManager.checkAuthorization(reverseClient, req.streamChunkId.streamId);
    streamManager.registerChannel(channel, req.streamChunkId.streamId);
    buf = streamManager.getChunk(req.streamChunkId.streamId, req.streamChunkId.
        chunkIndex);
  } catch (Exception e) {
    logger.error(String.format("Error opening block %s for request from %s",
      req.streamChunkId, getRemoteAddress(channel)), e);
    respond(new ChunkFetchFailure(req.streamChunkId, Throwables.
        getStackTraceAsString(e)));
    return;
  }
  respond(new ChunkFetchSuccess(req.streamChunkId, buf));
}
```

代码清单 3-29 中的 streamManager 是通过调用 RpcHandler 的 getStreamManager 方法获取的 StreamManager。processFetchRequest 的处理都依托于 RpcHandler 的 StreamManager，其处理步骤如下。

1）调用 StreamManager 的 checkAuthorization 方法，校验客户端是否有权限从给定的流中读取。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

2）调用 StreamManager 的 registerChannel 方法，将一个流和一条（只能是一条）客户端的 TCP 连接关联起来，这可以保证对于单个的流只会有一个客户端读取。流关闭之后就永远不能够重用了。

3）调用 StreamManager 的 getChunk 方法，获取单个的块（块被封装为 ManagedBuffer）。由于单个的流只能与单个的 TCP 连接相关联，因此 getChunk 方法不能为了某个特殊的流而并行调用。

4）将 ManagedBuffer 和流的块 Id 封装为 ChunkFetchSuccess 后，调用 respond 方法返回给客户端。

有关 StreamManager 的具体实现，读者可以参考 5.3.5 节介绍的 NettyStreamManager 和 6.9.2 节介绍的 NettyBlockRpcServer 中的 OneForOneStreamManager。

2. 处理 RPC 请求

processRpcRequest 方法用于处理 RpcRequest 类型的消息，其实现如代码清单 3-30 所示。

代码清单3-30 processRpcRequest的实现

```
private void processRpcRequest(final RpcRequest req) {
  try {
    rpcHandler.receive(reverseClient, req.body().nioByteBuffer(), new RpcResponse
        Callback() {
      @Override
      public void onSuccess(ByteBuffer response) {
        respond(new RpcResponse(req.requestId, new NioManagedBuffer(response)));
      }

      @Override
      public void onFailure(Throwable e) {
        respond(new RpcFailure(req.requestId, Throwables.getStackTraceAsString(e)));
      }
    });
  } catch (Exception e) {
    logger.error("Error while invoking RpcHandler#receive() on RPC id " + req.
        requestId, e);
    respond(new RpcFailure(req.requestId, Throwables.getStackTraceAsString(e)));
  } finally {
    req.body().release();
  }
}
```

代码清单 3-30 中将 RpcRequest 消息的内容体、发送消息的客户端及一个 RpcResponseCallback 类型的匿名内部类作为参数传递给了 RpcHandler 的 receive 方法。这就是说真正用于处理 RpcRequest 消息的是 RpcHandler，而非 TransportRequestHandler。由于 RpcHandler 是抽象类（见代码清单 3-27），其 receive 方法也是抽象方法，所以具体的操作将由 RpcHandler 的实现了 receive 方法的子类来完成。所有继承 RpcHandler 的子类都需要在其 receive 方法的具体实现中回调 RpcResponseCallback 的 onSuccess（处理成功时）或者

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

onFailure（处理失败时）方法。从 RpcResponseCallback 的实现来看，无论处理结果成功还是失败，都将调用 respond 方法对客户端进行响应。

3. 处理流请求

processStreamRequest 方法用于处理 StreamRequest 类型的消息，其实现如代码清单 3-31 所示。

代码清单3-31　processStreamRequest的实现

```
private void processStreamRequest(final StreamRequest req) {
  ManagedBuffer buf;
  try {
    buf = streamManager.openStream(req.streamId);// 将获取到的流数据封装为ManagedBuffer
  } catch (Exception e) {
    logger.error(String.format(
      "Error opening stream %s for request from %s", req.streamId,
          getRemoteAddress(channel)), e);
    respond(new StreamFailure(req.streamId, Throwables.getStackTrace-
        AsString(e)));
    return;
  }
  if (buf != null) {
    respond(new StreamResponse(req.streamId, buf.size(), buf));
  } else {
    respond(new StreamFailure(req.streamId, String.format(
      "Stream '%s' was not found.", req.streamId)));
  }
}
```



代码清单 3-31 中也使用了 RpcHandler 的 StreamManager，其处理步骤如下。

1）调用 StreamManager 的 openStream 方法将获取到的流数据封装为 ManagedBuffer。

2）当成功或失败时调用 respond 方法向客户端响应。

4. 处理无需回复的 RPC 请求

processOneWayMessage 方法用于处理 StreamRequest 类型的消息，其实现如代码清单 3-32 所示。

代码清单3-32　processOneWayMessage的实现

```
private void processOneWayMessage(OneWayMessage req) {
  try {
    rpcHandler.receive(reverseClient, req.body().nioByteBuffer());
  } catch (Exception e) {
    logger.error("Error while invoking RpcHandler#receive() for one-way
        message.", e);
  } finally {
    req.body().release();
  }
}
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

processOneWayMessage 方法的实现与 processRpcRequest 非常相似，区别在于 processOneWayMessage 调用了代码清单 3-27 中 ONE_WAY_CALLBACK 的 receive 方法，因而 processOneWayMessage 在处理完 RPC 请求后不会对客户端作出响应。

从以上 4 种处理的分析可以看出，最终的处理都由 RpcHandler 及其内部组件完成。除了 OneWayMessage 的消息外，其余三种消息都是最终调用 respond 方法响应客户端，其实现如代码清单 3-33 所示。

代码清单3-33 respond的实现

```
private void respond(final Encodable result) {
  final SocketAddress remoteAddress = channel.remoteAddress();
  channel.writeAndFlush(result).addListener(
    new ChannelFutureListener() {
      @Override
      public void operationComplete(ChannelFuture future) throws Exception {
        if (future.isSuccess()) {
          logger.trace("Sent result {} to client {}", result, remoteAddress);
        } else {
          logger.error(String.format("Error sending result %s to %s; closing
              connection",
            result, remoteAddress), future.cause());
          channel.close();
        }
      }
    }
  );
}
```

可以看到 respond 方法中实际调用了 Channel 的 writeAndFlush 方法[⊖]来响应客户端。

3.2.7 服务端引导程序 TransportServerBootstrap

TransportServer 的构造器（见代码清单 3-15）中的 bootstraps 是 TransportServerBootstrap 的列表。接口 TransportServerBootstrap 定义了服务端引导程序的规范，服务端引导程序旨在当客户端与服务端建立连接之后，在服务端持有的客户端管道上执行的引导程序。TransportServerBootstrap 的定义如代码清单 3-34 所示。

代码清单3-34 TransportServerBootstrap的定义

```
public interface TransportServerBootstrap {
  RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler);
}
```

TransportServerBootstrap 的 doBootstrap 方法将对服务端的 RpcHandler 进行代理，接收客户端的请求。TransportServerBootstrap 有 SaslServerBootstrap 和 EncryptionChecker-Bootstrap

⊖ Channel 的 writeAndFlush 方法涉及 Netty 的实现细节及原理，这并不是本书所要阐述的内容，有兴趣的读者可以访问 Netty 官网：http://netty.io 获取更多信息。

仅供非商业用途或交流学习使用

两个实现类。为了更清楚地说明 TransportServerBootstrap 的意义，我们以 SaslServerBootstrap 为例，来讲解其实现（见代码清单 3-35）。

代码清单3-35　SaslServerBootstrap的doBootstrap实现

```
public RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler) {
  return new SaslRpcHandler(conf, channel, rpcHandler, secretKeyHolder);
}
```

根据代码清单 3-35 我们知道，SaslServerBootstrap 的 doBootstrap 方法实际创建了 SaslRpcHandler，SaslRpcHandler 负责对管道进行 SASL（Simple Authentication and Security Layer）加密。SaslRpcHandler 本身也继承了 RpcHandler，所以我们重点来看其 receive 方法的实现，如代码清单 3-36 所示。

代码清单3-36　SaslRpcHandler的receive方法

```
@Override
public void receive(TransportClient client, ByteBuffer message, RpcResponseCallback
    callback) {
  if (isComplete) {
    // 将消息传递给SaslRpcHandler所代理的下游RpcHandler并返回
    delegate.receive(client, message, callback);
    return;
  }

  ByteBuf nettyBuf = Unpooled.wrappedBuffer(message);
  SaslMessage saslMessage;
  try {
    saslMessage = SaslMessage.decode(nettyBuf);// 对客户端发送的消息进行SASL解密
  } finally {
    nettyBuf.release();
  }

  if (saslServer == null) {
    // 如果saslServer还未创建，则需要创建SparkSaslServer
    client.setClientId(saslMessage.appId);
    saslServer = new SparkSaslServer(saslMessage.appId, secretKeyHolder,
      conf.saslServerAlwaysEncrypt());
  }
  byte[] response;
  try {
    response = saslServer.response(JavaUtils.bufferToArray(// 使用saslServer处理已
        解密的消息
      saslMessage.body().nioByteBuffer()));
  } catch (IOException ioe) {
    throw new RuntimeException(ioe);
  }
  callback.onSuccess(ByteBuffer.wrap(response));
  if (saslServer.isComplete()) {
    logger.debug("SASL authentication successful for channel {}", client);
    isComplete = true;// SASL认证交换已经完成
    if (SparkSaslServer.QOP_AUTH_CONF.equals(saslServer.getNegotiatedProperty(Sasl.
        QOP))) {
```

仅供非商业用途或交流学习使用


```
            logger.debug("Enabling encryption for channel {}", client);
            // 对管道进行SASL加密
            SaslEncryption.addToChannel(channel, saslServer, conf.maxSaslEncrypted-
                BlockSize());
            saslServer = null;
          } else {
            saslServer.dispose();
            saslServer = null;
          }
        }
      }
```

根据代码清单 3-36，SaslRpcHandler 处理客户端消息的步骤如下。

1）如果 SASL 认证交换已经完成（isComplete 等于 true），则将消息传递给 SaslRpcHandler 所代理的下游 RpcHandler 并返回。

2）如果 SASL 认证交换未完成（isComplete 等于 false），则对客户端发送的消息进行 SASL 解密。

3）如果 saslServer 还未创建，则需要创建 SparkSaslServer。当 SaslRpcHandler 接收到客户端的第一条消息时会做此操作。

4）使用 saslServer 处理已解密的消息，并将处理结果通过 RpcResponseCallback 的回调方法返回给客户端。

5）如果 SASL 认证交换已经完成，则将 isComplete 置为 true。

6）对管道进行 SASL 加密。

SaslServerBootstrap 是通过 SaslRpcHandler 对下游 RpcHandler 进行代理的一种 TransportServerBootstrap。EncryptionCheckerBootstrap 是另一种 TransportServerBootstrap 的实现，它通过将自身加入 Netty 的管道中实现引导，EncryptionCheckerBootstrap 的 doBootstrap 方法的实现如代码清单 3-37 所示。

代码清单3-37　EncryptionCheckerBootstrap的doBootstrap实现

```
@Override
public RpcHandler doBootstrap(Channel channel, RpcHandler rpcHandler) {
  channel.pipeline().addFirst("encryptionChecker", this);
  return rpcHandler;
}
```

在详细介绍了 TransportChannelHandler 之后，我们就可以对图 3-3 进行扩展，把 TransportRequestHandler、TransportServerBootstrap 及 RpcHandler 的处理流程增加进来，如图 3-7 所示。

有读者可能会问，图 3-7 中并未见 TransportServerBootstrap 的身影。根据对 TransportServerBootstrap 的两种实现的举例，我们知道 TransportServerBootstrap 将可能存在于图中任何两个组件的箭头连线中间，起到引导、包装、代理的作用。

仅供非商业用途或交流学习使用

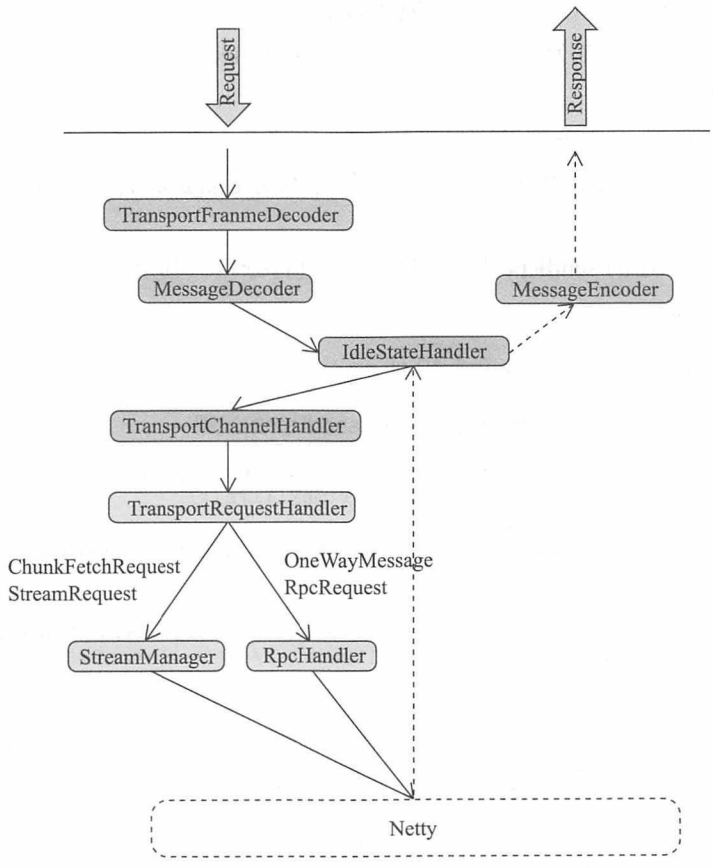


图 3-7　RPC 框架服务端处理请求、响应流程图

3.2.8　客户端 TransportClient 详解

在介绍完服务端 RpcHandler 对请求消息的处理之后，现在来看看客户端发送 RPC 请求的原理。我们在分析代码清单 3-18 中的 createChannelHandler 方法时，看到调用了 TransportClient 的构造器（见代码清单 3-38），其中 TransportResponseHandler 的引用将赋给 handler 属性。

代码清单3-38　TransportClient的构造器

```
public TransportClient(Channel channel, TransportResponseHandler handler) {
  this.channel = Preconditions.checkNotNull(channel);
  this.handler = Preconditions.checkNotNull(handler);
  this.timedOut = false;
}
```

TransportClient 一共有 5 个方法用于发送请求，分别如下。

1）fetchChunk：从远端协商好的流中请求单个块。

仅供非商业用途或交流学习使用

2）stream：使用流的 ID，从远端获取流数据。

3）sendRpc：向服务端发送 RPC 的请求，通过 At least Once Delivery 原则保证请求不会丢失。

4）sendRpcSync：向服务端发送异步的 RPC 的请求，并根据指定的超时时间等待响应。

5）send：向服务端发送 RPC 的请求，但是并不期望能获取响应，因而不能保证投递的可靠性。

本节只选择最常用的 sendRpc 和 fetchChunk 进行分析，其余实现都可以触类旁通。

1. 发送 RPC 请求

sendRpc 方法的实现如代码清单 3-39 所示。

代码清单3-39 sendRpc的实现

```
public long sendRpc(ByteBuffer message, final RpcResponseCallback callback) {
  final long startTime = System.currentTimeMillis();
  if (logger.isTraceEnabled()) {
    logger.trace("Sending RPC to {}", getRemoteAddress(channel));
  }
  // 使用UUID生成请求主键requestId
  final long requestId = Math.abs(UUID.randomUUID().getLeastSignificantBits());
  handler.addRpcRequest(requestId, callback);// 添加requestId与RpcResponseCallback
      的引用之间的关系
  // 发送RPC请求
  channel.writeAndFlush(new RpcRequest(requestId, new NioManagedBuffer
      (message))).addListener(
    new ChannelFutureListener() {
      @Override
      public void operationComplete(ChannelFuture future) throws Exception {
        if (future.isSuccess()) {
          long timeTaken = System.currentTimeMillis() - startTime;
          if (logger.isTraceEnabled()) {
            logger.trace("Sending request {} to {} took {} ms", requestId,
              getRemoteAddress(channel), timeTaken);
          }
        } else {
          String errorMsg = String.format("Failed to send RPC %s to %s: %s", requestId,
            getRemoteAddress(channel), future.cause());
          logger.error(errorMsg, future.cause());
          handler.removeRpcRequest(requestId);
          channel.close();
          try {
            callback.onFailure(new IOException(errorMsg, future.cause()));
          } catch (Exception e) {
            logger.error("Uncaught exception in RPC response callback handler!", e);
          }
        }
      }
    });
  return requestId;
}
```

仅供非商业用途或交流学习使用

结合代码清单 3-39，我们知道 sendRpc 方法的实现步骤如下。

1）使用 UUID 生成请求主键 requestId。

2）调用 addRpcRequest 向 handler（特别提醒下读者，这里的 handler 不是 RpcHandler，而是通过 TransportClient 构造器传入的 TransportResponseHandler）添加 requestId 与回调类 RpcResponseCallback 的引用之间的关系。TransportResponseHandler 的 addRpcRequest 方法（见代码清单 3-40）将更新最后一次请求的时间为当前系统时间，然后将 requestId 与 RpcResponseCallback 之间的映射加入到 outstandingRpcs 缓存中。outstandingRpcs 专门用于缓存发出的 RPC 请求信息。

代码清单3-40　添加RPC请求到缓存

```
public void addRpcRequest(long requestId, RpcResponseCallback callback) {
  updateTimeOfLastRequest();
  outstandingRpcs.put(requestId, callback);
}
```

3）调用 Channel 的 writeAndFlush 方法将 RPC 请求发送出去，这和在代码清单 3-33 中服务端调用的 respond 方法响应客户端的一样，都是使用 Channel 的 writeAndFlush 方法。当发送成功或者失败时会回调 ChannelFutureListener 的 operationComplete 方法。如果发送成功，那么只会打印 requestId、远端地址及花费时间的日志，如果发送失败，除了打印错误日志外，还要调用 TransportResponseHandler 的 removeRpcRequest 方法（见代码清单 3-41），将此次请求从 outstandingRpcs 缓存中移除。



代码清单3-41　从缓存中删除RPC请求

```
public void removeRpcRequest(long requestId) {
  outstandingRpcs.remove(requestId);
}
```

请求发送成功后，客户端将等待接收服务端的响应。根据图 3-3，返回的消息也会传递给 TransportChannelHandler 的 channelRead 方法（见代码清单 3-19），根据之前的分析，消息的分析将最后交给 TransportResponseHandler 的 handle 方法来处理。TransportResponseHandler 的 handle 方法分别对图 3-5 中的 6 种 ResponseMessage 进行处理，由于服务端使用 processRpcRequest 方法（见代码清单 3-30）处理 RpcRequest 类型的消息后，返回给客户端的消息为 RpcResponse 或 RpcFailure，所以我们来看看客户端的 TransportResponseHandler 的 handle 方法如何处理 RpcResponse 和 RpcFailure，如代码清单 3-42 所示。

代码清单3-42　RpcResponse和RpcFailure消息的处理

```
} else if (message instanceof RpcResponse) {
  RpcResponse resp = (RpcResponse) message;
  RpcResponseCallback listener = outstandingRpcs.get(resp.requestId);// 获取
      RpcResponseCallback
```

仅供非商业用途或交流学习使用


```
    if (listener == null) {
      logger.warn("Ignoring response for RPC {} from {} ({} bytes) since it is not
          outstanding",
        resp.requestId, getRemoteAddress(channel), resp.body().size());
    } else {
      outstandingRpcs.remove(resp.requestId);
      try {
        listener.onSuccess(resp.body().nioByteBuffer());
      } finally {
        resp.body().release();
      }
    }
  } else if (message instanceof RpcFailure) {
    RpcFailure resp = (RpcFailure) message;
    RpcResponseCallback listener = outstandingRpcs.get(resp.requestId); // 获取
        RpcResponseCallback
    if (listener == null) {
      logger.warn("Ignoring response for RPC {} from {} ({}) since it is not
          outstanding",
        resp.requestId, getRemoteAddress(channel), resp.errorString);
    } else {
      outstandingRpcs.remove(resp.requestId);
      listener.onFailure(new RuntimeException(resp.errorString));
    }
```

从代码清单 3-42 看到，处理 RpcResponse 的逻辑如下。

1）使用 RpcResponse 对应的 RpcRequest 的主键 requestId，从 outstandingRpcs 缓存中获取注册的 RpcResponseCallback，此处的 RpcResponseCallback 即为代码清单 3-39 中传递给 sendRpc 方法的 RpcResponseCallback。

2）移除 outstandingRpcs 缓存中 requestId 和 RpcResponseCallback 的注册信息。

3）调用 RpcResponseCallback 的 onSuccess 方法，处理成功响应后的具体逻辑。这里的 RpcResponseCallback 需要在各个使用 TransportClient 的 sendRpc 方法的场景中分别实现。

4）释放 RpcResponse 的 body，回收资源。

处理 RpcFailure 的逻辑如下。

1）使用 RpcFailure 对应的 RpcRequest 的主键 requestId，从 outstandingRpcs 缓存中获取注册的 RpcResponseCallback，此处的 RpcResponseCallback 即为代码清单 3-39 中传递给 sendRpc 方法的 RpcResponseCallback。

2）移除 outstandingRpcs 缓存中 requestId 和 RpcResponseCallback 的注册信息。

3）调用 RpcResponseCallback 的 onFailure 方法，处理失败响应后的具体逻辑。这里的 RpcResponseCallback 需要在使用 TransportClient 的 sendRpc 方法时指定或实现。

2. 发送获取块请求

fetchChunk 的实现如代码清单 3-43 所示。

仅供非商业用途或交流学习使用

代码清单3-43 fetchChunk的实现

```
public void fetchChunk(
    long streamId,
    final int chunkIndex,
    final ChunkReceivedCallback callback) {
  final long startTime = System.currentTimeMillis();
  if (logger.isDebugEnabled()) {
    logger.debug("Sending fetch chunk request {} to {}", chunkIndex,
        getRemoteAddress (channel));
  }
  final StreamChunkId streamChunkId = new StreamChunkId(streamId, chunkIndex);//
      创建StreamChunkId
  // 添加StreamChunkId与ChunkReceivedCallback之间的对应关系
  handler.addFetchRequest(streamChunkId, callback);
  // 发送块请求
  channel.writeAndFlush(new ChunkFetchRequest(streamChunkId)).addListener(
    new ChannelFutureListener() {
      @Override
      public void operationComplete(ChannelFuture future) throws Exception {
        if (future.isSuccess()) {
          long timeTaken = System.currentTimeMillis() - startTime;
          if (logger.isTraceEnabled()) {
            logger.trace("Sending request {} to {} took {} ms", streamChunkId,
              getRemoteAddress(channel), timeTaken);
          }
        } else {
          String errorMsg = String.format("Failed to send request %s to %s: %s",
              streamChunkId,
            getRemoteAddress(channel), future.cause());
          logger.error(errorMsg, future.cause());
          handler.removeFetchRequest(streamChunkId);
          channel.close();
          try {
            callback.onFailure(chunkIndex, new IOException(errorMsg, future.cause()));
          } catch (Exception e) {
            logger.error("Uncaught exception in RPC response callback handler!", e);
          }
        }
      }
    });
}
```

结合代码清单3-43，我们知道fetchChunk方法的实现步骤如下。

1）使用流的标记streamId和块的索引chunkIndex创建StreamChunkId。

2）调用addFetchRequest向handler（特别提醒下读者，这里的handler不是RpcHandler，而是通过TransportClient构造器传入的TransportResponseHandler）添加StreamChunkId与回调类ChunkReceivedCallback的引用之间的关系。TransportResponseHandler的addFetchRequest方法（见代码清单3-44）将更新最后一次请求的时间为当前系统时间，然后将StreamChunkId与ChunkReceivedCallback之间的映射加入到outstandingFetches缓存中。outstandingFetches专门用于缓存发出的块请求信息。

仅供非商业用途或交流学习使用

代码清单3-44 添加块请求到缓存

```
public void addFetchRequest(StreamChunkId streamChunkId, ChunkReceivedCallback
    callback) {
  updateTimeOfLastRequest();
  outstandingFetches.put(streamChunkId, callback);
}
```

3）调用 Channel 的 writeAndFlush 方法将**块请求**发送出去，这和在代码清单 3-33 中服务端调用的 respond 方法响应客户端的一样，都是使用 Channel 的 writeAndFlush 方法。当发送成功或者失败时会回调 ChannelFutureListener 的 operationComplete 方法。如果发送成功，那么只会打印 StreamChunkId、远端地址及花费时间的日志，如果发送失败，除了打印错误日志外，还要调用 TransportResponseHandler 的 removeFetchRequest 方法（见代码清单 3-45），将此次请求从 outstandingFetches 缓存中移除。

代码清单3-45 从缓存中删除RPC请求

```
public void removeRpcRequest(long requestId) {
  outstandingRpcs.remove(requestId);
}
```

请求发送成功后，客户端将等待接收服务端的响应。根据图 3-3，返回的消息也会传递给 TransportChannelHandler 的 channelRead 方法（见代码清单 3-19），根据之前的分析，消息的分析将最后交给 TransportResponseHandler 的 handle 方法来处理。TransportResponseHandler 的 handle 方法分别对图 3-5 中的 6 种处理结果进行处理，由于服务端使用 processFetchRequest 方法（见代码清单 3-29）处理 ChunkFetchRequest 类型的消息后返回给客户端的消息为 ChunkFetchSuccess 或 ChunkFetchFailure，所以我们来看看客户端的 TransportResponseHandler 的 handle 方法如何处理 ChunkFetchSuccess 和 ChunkFetchFailure，如代码清单 3-46 所示。



代码清单3-46 ChunkFetchSuccess和ChunkFetchFailure消息的处理

```
if (message instanceof ChunkFetchSuccess) {
  ChunkFetchSuccess resp = (ChunkFetchSuccess) message;
  ChunkReceivedCallback listener = outstandingFetches.get(resp.streamChunkId);
  if (listener == null) {
    logger.warn("Ignoring response for block {} from {} since it is not
        outstanding",
      resp.streamChunkId, getRemoteAddress(channel));
    resp.body().release();
  } else {
    outstandingFetches.remove(resp.streamChunkId);
    listener.onSuccess(resp.streamChunkId.chunkIndex, resp.body());
    resp.body().release();
  }
} else if (message instanceof ChunkFetchFailure) {
  ChunkFetchFailure resp = (ChunkFetchFailure) message;
  ChunkReceivedCallback listener = outstandingFetches.get(resp.streamChunkId);
```

仅供非商业用途或交流学习使用


```
    if (listener == null) {
      logger.warn("Ignoring response for block {} from {} ({}) since it is not
          outstanding",
        resp.streamChunkId, getRemoteAddress(channel), resp.errorString);
    } else {
      outstandingFetches.remove(resp.streamChunkId);
      listener.onFailure(resp.streamChunkId.chunkIndex, new ChunkFetchFailure-
          Exception(
        "Failure while fetching " + resp.streamChunkId + ": " + resp.errorString));
    }
  }
```

从代码清单3-46看到，处理ChunkFetchSuccess的逻辑如下。

1）使用ChunkFetchSuccess对应的StreamChunkId，从outstandingFetches缓存中获取注册的ChunkReceivedCallback，此处的ChunkReceivedCallback即为代码清单3-43中传递给fetchChunk方法的ChunkReceivedCallback。

2）移除outstandingFetches缓存中StreamChunkId和ChunkReceivedCallback的注册信息。

3）调用ChunkReceivedCallback的onSuccess方法，处理成功响应后的具体逻辑。这里的ChunkReceivedCallback需要在各个使用TransportClient的fetchChunk方法的场景中分别实现。

4）释放ChunkFetchSuccess的body，回收资源。

处理ChunkFetchFailure的逻辑如下。

1）使用ChunkFetchFailure对应的StreamChunkId，从outstandingFetches缓存中获取注册的ChunkReceivedCallback，此处的ChunkReceivedCallback即为代码清单3-43中传递给fetchChunk方法的ChunkReceivedCallback。

2）移除outstandingFetches缓存中StreamChunkId和ChunkReceivedCallback的注册信息。

3）调用ChunkReceivedCallback的onFailure方法，处理失败响应后的具体逻辑。这里的ChunkReceivedCallback需要在各个使用TransportClient的fetchChunk方法的场景中分别实现。

在详细介绍了TransportClient和TransportResponseHandler之后，对于客户端我们就可以扩展图3-3，把TransportResponseHandler及TransportClient的处理流程增加进来，如图3-8所示。

图3-8中的序号①表示调用TransportResponseHandler的addRpcRequest方法（或addFetchRequest方法），将更新最后一次请求的时间为当前系统时间，然后将requestId与RpcResponseCallback之间的映射加入到outstandingRpcs缓存中（或将StreamChunkId与ChunkReceivedCallback之间的映射加入到outstandingFetches缓存中）。序号②表示调用Channel的writeAndFlush方法将RPC请求发送出去。图中的虚线表示当TransportResponseHandler处理RpcResponse和RpcFailure时，将从outstandingRpcs缓存中获取此请求对应的RpcResponseCallback（或处理ChunkFetchSuccess和ChunkFetchFailure时，将从outstandingFetches缓存中获取StreamChunkId对应的ChunkReceivedCallback），并执行回调。

仅供非商业用途或交流学习使用

此外，TransportClientBootstrap 将可能存在于图 3-8 中任何两个组件的箭头连线中间。

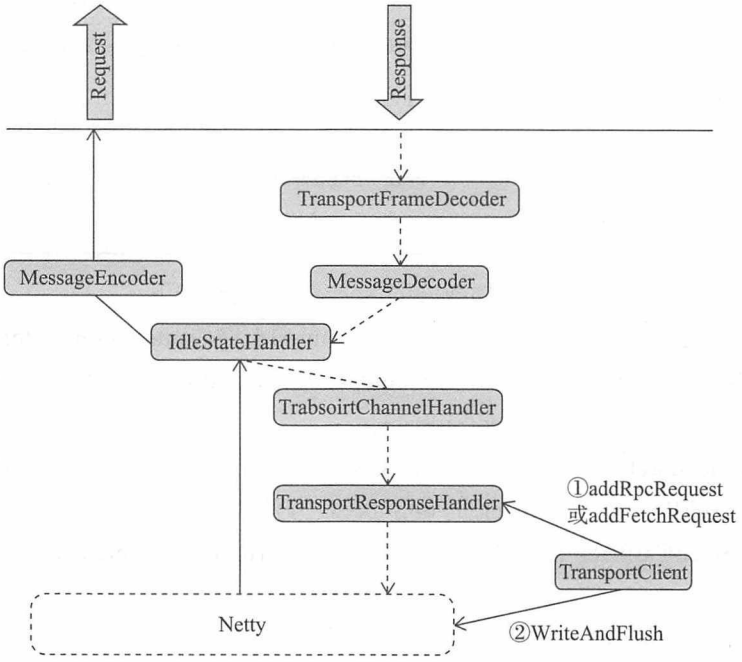


图 3-8 客户端请求、响应流程图

3.3 事件总线

Spark 定义了一个特质[⊖] ListenerBus，可以接收事件并且将事件提交到对应事件的监听器。为了对 ListenerBus 有个直观的理解，我们先来看看它的代码实现，如代码清单 3-47 所示。

代码清单3-47 ListenerBus的定义

```
private[spark] trait ListenerBus[L <: AnyRef, E] extends Logging {

private[spark] val listeners = new CopyOnWriteArrayList[L]

final def addListener(listener: L): Unit = {
  listeners.add(listener)
}

final def removeListener(listener: L): Unit = {
  listeners.remove(listener)
}
```

⊖ 特质是 Scala 语言中提供真正的多重继承的语法特性，类似于 Java 中的 Interface，但是又可以实现方法。有关 Scala 特质的更多介绍，请访问 Scala 官网 http://www.scala-lang.org。

仅供非商业用途或交流学习使用


```
final def postToAll(event: E): Unit = {
  val iter = listeners.iterator
  while (iter.hasNext) {
    val listener = iter.next()
    try {
      doPostEvent(listener, event)
    } catch {
      case NonFatal(e) =>
        logError(s"Listener ${Utils.getFormattedClassName(listener)} threw an
            exception", e)
    }
  }
}

protected def doPostEvent(listener: L, event: E): Unit

private[spark] def findListenersByClass[T <: L : ClassTag](): Seq[T] = {
  val c = implicitly[ClassTag[T]].runtimeClass
  listeners.asScala.filter(_.getClass == c).map(_.asInstanceOf[T]).toSeq
}

}
```

代码清单 3-47 中展示了 ListenerBus 是个泛型特质，其泛型参数为 [L <: AnyRef, E]，其中 L 是代表监听器的泛型参数，可以看到 ListenerBus 支持任何类型的监听器，E 是代表事件的泛型参数。ListenerBus 中各个成员的作用如下。

- listeners：用于维护所有注册的监听器，其数据结构为 CopyOnWriteArrayList[L]。
- addListener：向 listeners 中添加监听器的方法，由于 listeners 采用 CopyOnWriteArrayList 来实现，所以 addListener 方法是线程安全的。
- removeListener：从 listeners 中移除监听器的方法，由于 listeners 采用 CopyOnWriteArrayList 来实现，所以 removeListener 方法是线程安全的。
- postToAll：此方法的作用是将事件投递给所有的监听器。虽然 CopyOnWriteArrayList 本身是线程安全的，但是由于 postToAll 方法内部引入了“先检查后执行”的逻辑，因而 postToAll 方法不是线程安全的，所以所有对 postToAll 方法的调用应当保证在同一个线程中。
- doPostEvent：用于将事件投递给指定的监听器，此方法只提供了接口定义，具体实现需要子类提供。
- findListenersByClass：查找与指定类型相同的监听器列表。

3.3.1　ListenerBus 的继承体系

理解了 ListenerBus 的定义后，本节一起来看看有哪些类继承了它。ListenerBus 的类继承体系如图 3-9 所示。

从图 3-9 中可以看到有 3 种 ListenerBus 的具体实现，分别如下。

仅供非商业用途或交流学习使用

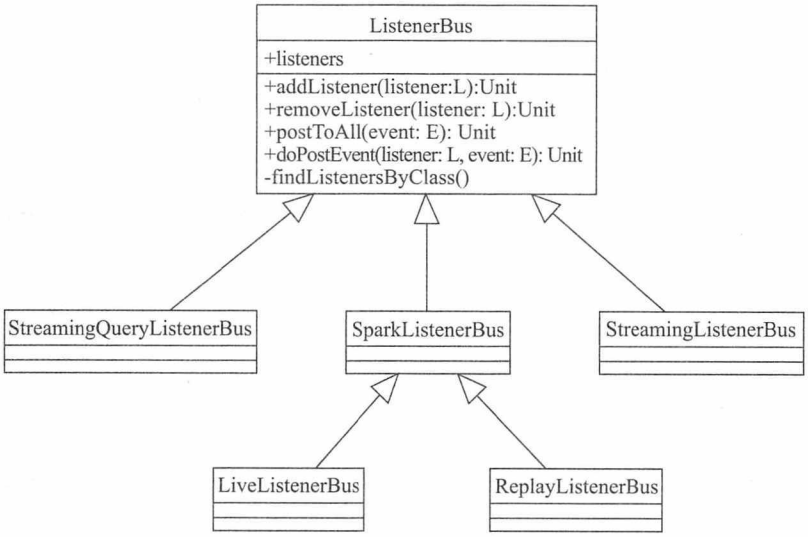


图 3-9 ListenerBus 的类继承体系

- SparkListenerBus：用于将 SparkListenerEvent 类型的事件投递到 SparkListener-Interface 类型的监听器。
- StreamingQueryListenerBus：用于将 StreamingQueryListener.Event 类型的事件投递到 StreamingQueryListener 类型的监听器，此外还会将 StreamingQueryListener.Event 类型的事件交给 SparkListenerBus。
- StreamingListenerBus：用于将 StreamingListenerEvent 类型的事件投递到 Streaming Listener 类型的监听器，此外还会将 StreamingListenerEvent 类型的事件交给 Spark ListenerBus。

SparkListenerBus 也有如下两种实现。

- LiveListenerBus：采用异步线程将 SparkListenerEvent 类型的事件投递到 SparkListener 类型的监听器。
- ReplayListenerBus：用于从序列化的事件数据中重播事件。

有了对事件总线的这些介绍，读者已经在宏观上对其有所认识，但是如果没有具体的实现，ListenerBus 本身也无法发挥作用。下一小节我们将针对 SparkListenerBus 从更加微观的角度说明如何使用事件总线。

3.3.2 SparkListenerBus 详解

有了之前对 ListenerBus 的定义及 ListenerBus 类继承体系的准备，本小节将详细介绍 SparkListenerBus 的实现，如代码清单 3-48 所示。⊖

⊖ 鉴于 SparkListenerBus 的实现在本书其他部分将多次用到，因而将其完整列出。

仅供非商业用途或交流学习使用

代码清单3-48　SparkListenerBus的实现

```
private[spark] trait SparkListenerBus
extends ListenerBus[SparkListenerInterface, SparkListenerEvent] {
protected override def doPostEvent(
    listener: SparkListenerInterface,
    event: SparkListenerEvent): Unit = {
  event match {
    case stageSubmitted: SparkListenerStageSubmitted =>
      listener.onStageSubmitted(stageSubmitted)
    case stageCompleted: SparkListenerStageCompleted =>
      listener.onStageCompleted(stageCompleted)
    case jobStart: SparkListenerJobStart =>
      listener.onJobStart(jobStart)
    case jobEnd: SparkListenerJobEnd =>
      listener.onJobEnd(jobEnd)
    case taskStart: SparkListenerTaskStart =>
      listener.onTaskStart(taskStart)
    case taskGettingResult: SparkListenerTaskGettingResult =>
      listener.onTaskGettingResult(taskGettingResult)
    case taskEnd: SparkListenerTaskEnd =>
      listener.onTaskEnd(taskEnd)
    case environmentUpdate: SparkListenerEnvironmentUpdate =>
      listener.onEnvironmentUpdate(environmentUpdate)
    case blockManagerAdded: SparkListenerBlockManagerAdded =>
      listener.onBlockManagerAdded(blockManagerAdded)
    case blockManagerRemoved: SparkListenerBlockManagerRemoved =>
      listener.onBlockManagerRemoved(blockManagerRemoved)
    case unpersistRDD: SparkListenerUnpersistRDD =>
      listener.onUnpersistRDD(unpersistRDD)
    case applicationStart: SparkListenerApplicationStart =>
      listener.onApplicationStart(applicationStart)
    case applicationEnd: SparkListenerApplicationEnd =>
      listener.onApplicationEnd(applicationEnd)
    case metricsUpdate: SparkListenerExecutorMetricsUpdate =>
      listener.onExecutorMetricsUpdate(metricsUpdate)
    case executorAdded: SparkListenerExecutorAdded =>
      listener.onExecutorAdded(executorAdded)
    case executorRemoved: SparkListenerExecutorRemoved =>
      listener.onExecutorRemoved(executorRemoved)
    case blockUpdated: SparkListenerBlockUpdated =>
      listener.onBlockUpdated(blockUpdated)
    case logStart: SparkListenerLogStart => // ignore event log metadata
    case _ => listener.onOtherEvent(event)
  }
}
}
```



我们看到 SparkListenerBus 已经实现了 ListenerBus 的 doPostEvent 方法，通过对 SparkListenerEvent 事件的匹配，执行 SparkListenerInterface 监听器的相应方法。

这里的 SparkListenerEvent 其实是个特质，代码清单 3-48 中列出的 SparkListener-

仅供非商业用途或交流学习使用

StageSubmitted、SparkListenerStageCompleted 等都是继承了 SparkListenerEvent 特质的样例类[⊖]。为了说明问题，这里仅仅摘选 SparkListenerEvent 及部分 SparkListenerEvent 子类，其实现如下。

```
@DeveloperApi
@JsonTypeInfo(use = JsonTypeInfo.Id.CLASS, include = JsonTypeInfo.As.PROPERTY,
    property = "Event")
trait SparkListenerEvent {
  protected[spark] def logEvent: Boolean = true
}
@DeveloperApi
case class SparkListenerStageSubmitted(stageInfo: StageInfo, properties: Properties =
                                       null)
  extends SparkListenerEvent
@DeveloperApi
case class SparkListenerStageCompleted(stageInfo: StageInfo)
  extends SparkListenerEvent
@DeveloperApi
case class SparkListenerTaskStart(stageId: Int, stageAttemptId: Int, taskInfo:
    TaskInfo)
  extends SparkListenerEvent
// 省略其他SparkListenerEvent的实现
private[spark] case class SparkListenerLogStart(sparkVersion: String) extends
    SparkListenerEvent
```

SparkListenerInterface 也是一个特质，其中定义了所有 SparkListener 应当遵守的接口规范。由于 SparkListenerInterface 中定义了很多接口，为说明问题，只摘抄 SparkListenerInterface 中的部分接口定义，其代码如下。

```
private[spark] trait SparkListenerInterface {
  def onStageCompleted(stageCompleted: SparkListenerStageCompleted): Unit
  def onStageSubmitted(stageSubmitted: SparkListenerStageSubmitted): Unit
  // 省略其他接口方法
  def onOtherEvent(event: SparkListenerEvent): Unit
}
```

结合代码清单 3-48，我们知道以上代码片段中的 onStageCompleted 和 onStageSubmitted 将在 SparkListenerBus 的 doPostEvent 方法中分别匹配到 SparkListenerStageCompleted 和 SparkListenerStageSubmitted 事件时执行，而对于 doPostEvent 中无法匹配的事件，都将执行 onOtherEvent 方法。

在详细介绍了 ListenerBus 及 SparkListenerBus 后，我们知道当有事件需要通知监听器的时候，可以调用 ListenerBus 的 postToAll 方法，postToAll 方法遍历所有监听器并调用 SparkListenerBus 实现的 doPostEvent 方法，doPostEvent 方法对事件类型进行匹配后调用监听器的不同方法。整个投递事件的过程是通过方法调用实现的，所以这是一个同步调用。在监听器比较多的时候，这个过程会相对比较耗时，在 Spark UI（将在第 4 章中详细介绍）

⊖ 样例类是 Scala 语言的语法特性，样例类是一种特殊的类型，常用作事件、参数、模式匹配等。有关样例类的更多介绍，请读者阅读 Scala 语言的相关资料。

仅供非商业用途或交流学习使用

中为了达到页面的即时刷新，实现了 SparkListenerBus 的子类 LiveListenerBus。下一小节将围绕 LiveListenerBus 来详细说明异步投递消息的实现细节。

3.3.3　LiveListenerBus 详解

LiveListenerBus 继承了 SparkListenerBus，并实现了将事件异步投递给监听器，达到实时刷新 UI 界面数据的效果。LiveListenerBus 主要由以下部分组成。

- eventQueue：是 SparkListenerEvent 事件的阻塞队列，队列大小可以通过 Spark 属性 spark.scheduler.listenerbus.eventqueue.size 进行配置，默认为 10000（Spark 早期版本中属于静态属性，固定为 10000）。
- started：标记 LiveListenerBus 的启动状态的 AtomicBoolean 类型的变量。
- stopped：标记 LiveListenerBus 的停止状态的 AtomicBoolean 类型的变量。
- droppedEventsCounter：使用 AtomicLong 类型对删除的事件进行计数，每当日志打印了 droppedEventsCounter 后，会将 droppedEventsCounter 重置为 0。
- lastReportTimestamp：用于记录最后一次日志打印 droppedEventsCounter 的时间戳。
- processingEvent：用来标记当前正有事件被 listenerThread 线程处理。
- logDroppedEvent：AtomicBoolean 类型的变量，用于标记是否由于 eventQueue 已满，导致新的事件被删除。
- eventLock：用于当有新的事件到来时释放信号量，当对事件进行处理时获取信号量。
- listeners：继承自 LiveListenerBus 的监听器数组。
- listenerThread：处理事件的线程。



1. 异步事件处理线程

listenerThread 用于异步处理 eventQueue 中的事件，为了便于说明，这里将展示 listenerThread 及 LiveListenerBus 中的主要代码片段，如代码清单 3-49 所示。

代码清单3-49　LiveListenerBus主要逻辑的代码片段

```
private lazy val EVENT_QUEUE_CAPACITY = validateAndGetQueueSize()
private lazy val eventQueue = new LinkedBlockingQueue[SparkListenerEvent](EVENT_
    QUEUE_CAPACITY)
private def validateAndGetQueueSize(): Int = {
  val queueSize = sparkContext.conf.get(LISTENER_BUS_EVENT_QUEUE_SIZE)
  if (queueSize <= 0) {
    throw new SparkException("spark.scheduler.listenerbus.eventqueue.size must be > 0!")
  }
  queueSize
}

private val started = new AtomicBoolean(false)
private val stopped = new AtomicBoolean(false)
private val droppedEventsCounter = new AtomicLong(0L)
@volatile private var lastReportTimestamp = 0L
private var processingEvent = false
```

仅供非商业用途或交流学习使用


```
private val logDroppedEvent = new AtomicBoolean(false)
private val eventLock = new Semaphore(0)

private val listenerThread = new Thread(name) {
  setDaemon(true)
  override def run(): Unit = Utils.tryOrStopSparkContext(sparkContext) {
    LiveListenerBus.withinListenerThread.withValue(true) {
      while (true) {
        eventLock.acquire() // 获取信号量
        self.synchronized {
          processingEvent = true
        }
        try {
          val event = eventQueue.poll //从eventQueue中获取事件
          if (event == null) {
            // Get out of the while loop and shutdown the daemon thread
            if (!stopped.get) {
              throw new IllegalStateException("Polling 'null' from eventQueue means" +
                " the listener bus has been stopped. So 'stopped' must be true")
            }
            return
          }
          postToAll(event) // 事件处理
        } finally {
          self.synchronized {
            processingEvent = false
          }
        }
      }
    }
  }
}
```



通过分析代码清单 3-49，listenerThread 的工作步骤如下。

1）不断获取信号量（当可以获取信号量时，说明还有事件未处理）。

2）通过同步控制，将 processingEvent 设置为 true。

3）从 eventQueue 中获取事件。

4）调用超类 ListenerBus 的 postToAll 方法（postToAll 方法对监听器进行遍历，并调用 SparkListenerBus 的 doPostEvent 方法对事件进行匹配后执行监听器的相应方法）。

5）每次循环结束依然需要通过同步控制，将 processingEvent 设置为 false。

值得一提的是，listenerThread 的 run 方法中调用了 Utils 的 tryOrStopSparkContext，tryOrStopSparkContext 方法可以保证当 listenerThread 的内部循环抛出异常后启动一个新的线程停止 SparkContext（SparkContext 的内容将在第 4 章详细介绍，tryOrStopSparkContext 方法的具体实现请阅读附录 A）。

2. LiveListenerBus 的消息投递

在解释了异步线程 listenerThread 的工作内容后，还有一个要点没有解释：eventQueue 中的事件是如何放进去的呢？由于 eventQueue 定义在 LiveListenerBus 中，因此 ListenerBus

書籍是人类进步的阶梯

仅供非商业用途或交流学习使用

和 SparkListenerBus 中并没有操纵 eventQueue 的方法，要将事件放入 eventQueue，只能依靠 LiveListenerBus 自己了，其 post 方法就是为此目的而生的，如代码清单 3-50 所示。

代码清单3-50　向LiveListenerBus投递SparkListenerEvent事件

```
def post(event: SparkListenerEvent): Unit = {
  if (stopped.get) {
    logError(s"$name has already stopped! Dropping event $event")
    return
  }
  val eventAdded = eventQueue.offer(event) // 向eventQueue中添加事件
  if (eventAdded) {
    eventLock.release()
  } else {
    onDropEvent(event)
    droppedEventsCounter.incrementAndGet()
  }
  // 打印删除事件数的日志
  val droppedEvents = droppedEventsCounter.get
  if (droppedEvents > 0) {
    if (System.currentTimeMillis() - lastReportTimestamp >= 60 * 1000) {
      if (droppedEventsCounter.compareAndSet(droppedEvents, 0)) {
        val prevLastReportTimestamp = lastReportTimestamp
        lastReportTimestamp = System.currentTimeMillis()
        logWarning(s"Dropped $droppedEvents SparkListenerEvents since " +
          new java.util.Date(prevLastReportTimestamp))
      }
    }
  }
}
```

从代码清单 3-50 看到 post 方法的处理步骤如下。

1）判断 LiveListenerBus 是否已经处于停止状态。

2）向 eventQueue 中添加事件。如果添加成功，则释放信号量，进而催化 listenerThread 有效工作。如果 eventQueue 已满造成添加失败，则移除事件，并对删除事件计数器 droppedEventsCounter 进行自增。

3）如果有事件被删除，并且当前系统时间距离上一次打印 droppedEventsCounter 超过了 60 秒，则将 droppedEventsCounter 打印到日志。

3. LiveListenerBus 与监听器

与 LiveListenerBus 配合使用的监听器，并非是父类 SparkListenerBus 的类型参数 SparkListenerInterface，而是继承自 SparkListenerInterface 的 SparkListener 及其子类。图 3-10 列出了 Spark 中监听器 SparkListener 及它的 6 种最常用的实现[⊖]。

SparkListener 虽然实现了 SparkListenerInterface 中的每个方法，但是其实都是空实现，具体的实现需要交给子类去完成。

⊖ 除了本节列出的 6 种 SparkListener 的子类外，还有很多其他的子类，这里就不一一列出了，感兴趣的读者可以查阅 Spark 相关文档或阅读源码知晓。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

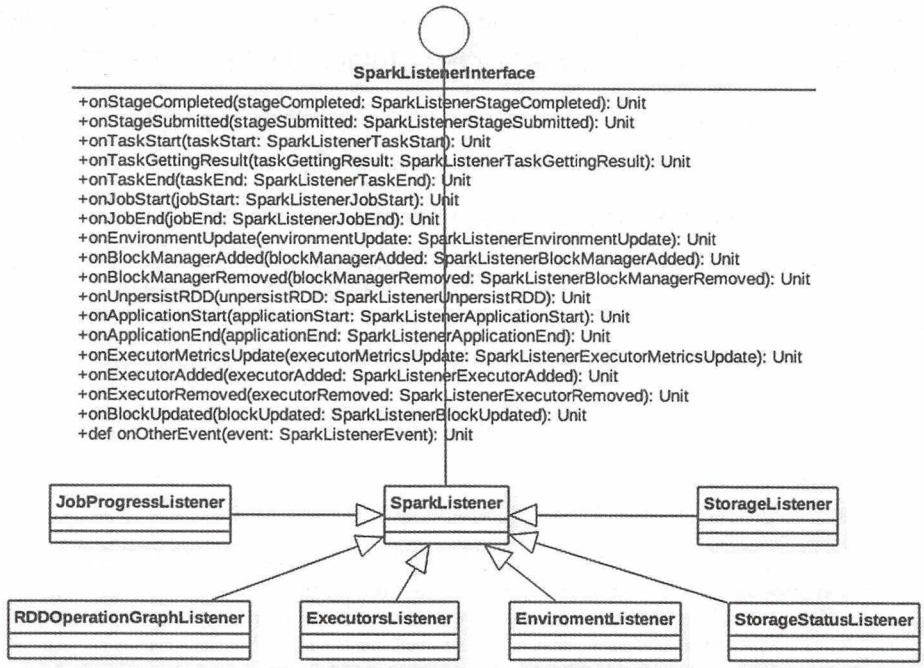


图 3-10 SparkListener 的类继承体系

本节首先对事件总线的接口定义进行了一些介绍，之后选择 ListenerBus 的子类 SparkListenerBus 与 LiveListenerBus 作为具体的实现例子进行分析，这里将通过图 3-11 更加直观地展示 ListenerBus、SparkListenerBus 及 LiveListenerBus 的工作原理。

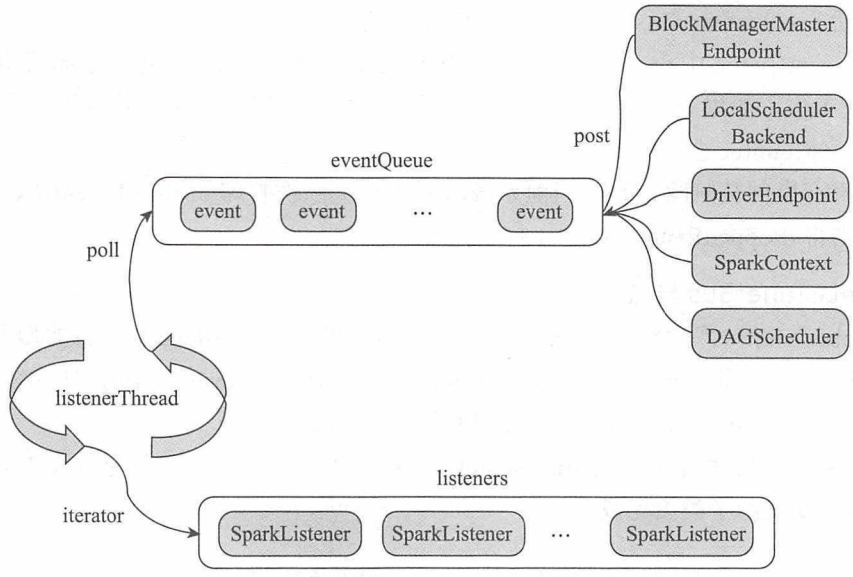


图 3-11 LiveListenerBus 的工作流程图

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

最后对图 3-11 作一些补充说明：图中的 DAGScheduler、SparkContext、BlockManagerMasterEndpoint、DriverEndpoint 及 LocalSchedulerBackend 都是 LiveListenerBus 的事件来源，它们都是通过调用 LiveListenerBus 的 post 方法将消息交给异步线程 listenerThread 进行处理的。

3.4　度量系统

对于一个系统而言，首先考虑要满足一些业务场景，并实现功能。随着系统功能越来越多，代码量级越来越高，系统的可维护性、可测试性、性能都会成为新的挑战，这使监控功能就变得越来越重要了。在国内，绝大多数 IT 公司的项目都以业务为导向，以完成功能为目标，这些项目在立项、设计、开发、上线的各个阶段，很少有人会考虑到监控的问题。开发人员能够认真地在代码段落中打印日志，就已经属于比较优秀的程序员了。然而，在国外很多项目则不会这样，看看久负盛名的 Hadoop 的监控系统就可见一斑，尤其是在 Facebook 中，更是把功能、日志及监控列为同等重要，这是作为一个合格工程师的“三驾马车”。

Spark 作为优秀的开源系统，在监控方面也有自己的一整套体系。一个系统有了监控功能后将收获诸多益处，如可测试性、性能优化、运维评估、数据统计等。Spark 的度量系统使用 codahale 提供的第三方度量仓库 Metrics，本节将着重介绍 Spark 基于 Metrics 构建度量系统的原理与实现。对于 Metrics 感兴趣的读者，可以参考阅读附录 D 中的内容。



Spark 的度量系统中有三个概念，分别如下。

- Instance：指定了度量系统的实例名。Spark 按照 Instance 的不同，区分为 Master、Worker、Application、Driver 和 Executor。
- Source：指定了从哪里收集度量数据，即度量数据的来源。Spark 提供了应用的度量来源（ApplicationSource）、Worker 的度量来源（WorkerSource）、DAGScheduler 的度量来源（DAGSchedulerSource）、BlockManager 的度量来源（BlockManagerSource）等诸多实现，对各个服务或组件进行监控。
- Sink：指定了往哪里输出度量数据，即度量数据的输出。Spark 中使用 MetricsServlet 作为默认的 Sink，此外还提供了 ConsoleSink、CsvSink、JmxSink、MetricsServlet、GraphiteSink 等实现。

为了更加直观地表现上述概念，我们以图 3-12 来表示 Spark 中度量系统的工作流程。

3.4.1　Source 继承体系

任何监控都离不开度量数据的采集，离线的数据采集很容易做到和被采集模块之间的解耦，但是对于实时的度量数据，尤其是那些内存中数据的采集就很难解耦。这就类似于网页监控数据的埋点一样，你要在网页中加入一段额外的 js 代码（如 Google 分析，即便你

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

只是引入一个 JS 文件，也很难让前端工程师感到开心）。还有一类监控，比如在 Java Web 中增加一个负责监控的 Servlet 或者一个基于 Spring 3.0 的拦截器，这种方式虽然将耦合度从代码级别降低到配置级别，但却无法有效地对内存中的数据结构进行监控。Spark 的度量系统对系统功能来说是在代码层面耦合的，这种牺牲对于能够换取对实时的、处于内存中的数据进行更有效的监控是值得的。

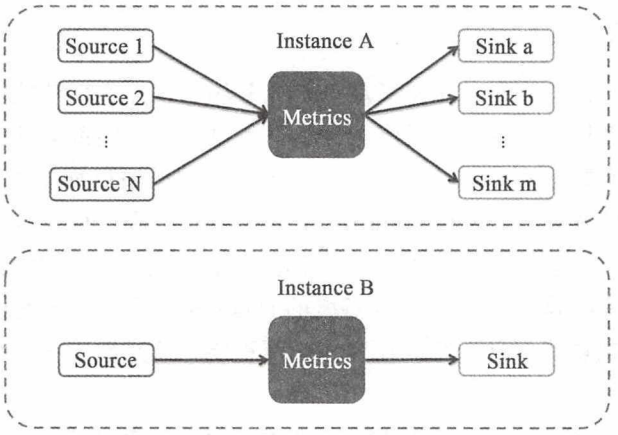


图 3-12 度量系统的工作流程

Spark 将度量来源抽象为 Source，其定义如代码清单 3-51 所示。

代码清单3-51 度量源的定义

```
private[spark] trait Source {
  def sourceName: String
  def metricRegistry: MetricRegistry
}
```

从代码清单 3-51 可以看到，Source 是一个特质，其中定义了两个方法。

- sourceName：度量源的名称。
- metricRegistry：当前度量源的注册表。MetricRegistry 是 Metrics 库提供的 API，在附录 D 中有更详细的介绍。

Spark 中有很多 Source 的具体实现，可以通过图 3-13 来了解。

为了说明 Source 该如何实现，我们选择 ApplicationSource（也是因为其实现简单明了，足以说明问题），其实现如代码清单 3-52 所示。

代码清单3-52 ApplicationSource的实现

```
private[master] class ApplicationSource(val application: ApplicationInfo) extends
Source {
override val metricRegistry = new MetricRegistry()
override val sourceName = "%s.%s.%s".format("application", application.desc.name,
  System.currentTimeMillis())
metricRegistry.register(MetricRegistry.name("status"), new Gauge[String] {
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    override def getValue: String = application.state.toString
  })
    metricRegistry.register(MetricRegistry.name("runtime_ms"), new Gauge[Long] {
      override def getValue: Long = application.duration
    })
    metricRegistry.register(MetricRegistry.name("cores"), new Gauge[Int] {
      override def getValue: Int = application.coresGranted
    })
  }
```

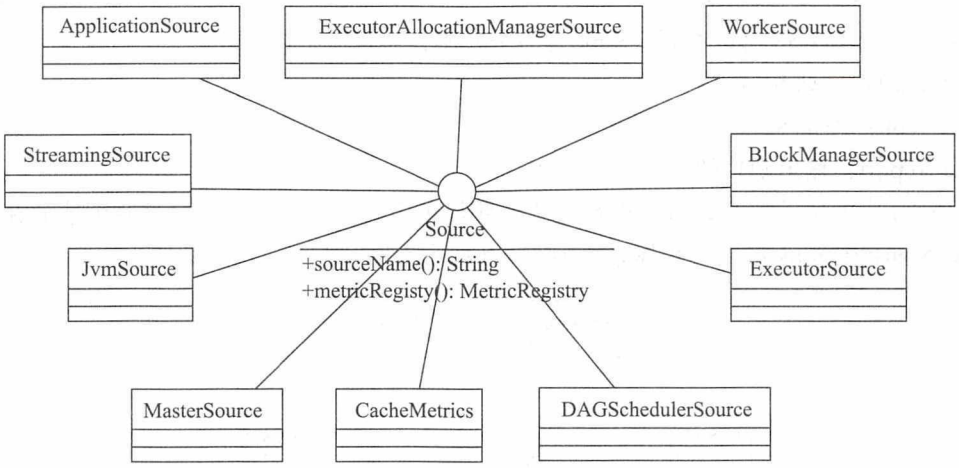


图 3-13　Source 的继承体系

望文生义，ApplicationSource 用于采集 Spark 应用程序相关的度量。代码清单 3-52 中 ApplicationSource 重载了 metricRegistry 和 sourceName，并且向自身的注册表注册了 status（即应用状态，包括 WAITING，RUNNING，FINISHED，FAILED，KILLED，UNKNOWN）、runtime_ms（运行持续时长）、cores（授权的内核数）等度量。这三个度量的取值分别来自于 ApplicationInfo 的 state、duration 和 coresGranted 三个属性。这三个度量都由 Gauge 的匿名内部类实现，Gauge 是 Metrics 提供的用于估计度量值的特质。有关 Gauge、MetricRegistry、MetricRegistry 注册度量的方法 register 及命名方法 name 的更详细介绍，请阅读附录 D。

3.4.2　Sink 继承体系

Source 准备好度量数据后，我们就需要考虑如何输出和使用的问题。这里介绍一些常见的度量输出方式：阿里数据部门采用的一种度量使用方式是输出到日志；在命令行运行过 Hadoop 任务（如 mapreduce）的使用者会发现控制台打印的内容中也包含度量信息；用户可能希望将有些度量信息保存到文件（如 CSV），以便将来能够查看；如果觉得使用 CSV 或者控制台等方式不够直观，还可以将采集到的度量数据输出到专用的监控系统界面。这

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

些最终对度量数据的使用，或者说是输出方式，Spark 将它们统一抽象为 Sink。Sink 的定义如代码清单 3-53 所示。

代码清单3-53 度量输出的定义

```
private[spark] trait Sink {
  def start(): Unit
  def stop(): Unit
  def report(): Unit
}
```

从代码清单 3-53 可以看到，Sink 是一个特质，包含三个接口方法。

- start：启动 Sink。
- stop：停止 Sink。
- report：输出到目的地。

从这三个方法的解释来看，很难让读者获得更多的信息。我们先把这些困惑放在一边，来看看 Spark 中 Sink 的类继承体系，如图 3-14 所示。

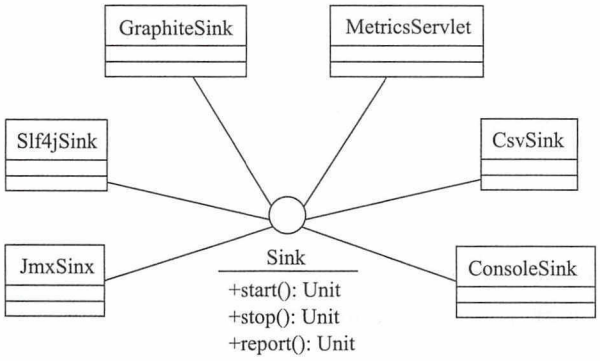


图 3-14 Sink 的类继承体系

图 3-14 中展示了 6 种 Sink 的具体实现。

- ConsoleSink：借助 Metrics 提供的 ConsoleReporter 的 API，将度量输出到 System.out，因此可以输出到控制台。
- CsvSink：借助 Metrics 提供的 CsvReporter 的 API，将度量输出到 CSV 文件。
- MetricsServlet：在 Spark UI 的 jetty 服务中创建 ServletContextHandler，将度量数据通过 Spark UI 展示在浏览器中。
- JmxSink：借助 Metrics 提供的 JmxReporter 的 API，将度量输出到 MBean 中，这样就可以打开 Java VisualVM，然后打开 Tomcat 进程监控，给 VisualVM 安装 MBeans 插件后，选择 MBeans 标签页可以对 JmxSink 所有注册到 JMX 中的对象进行管理。
- Slf4jSink：借助 Metrics 提供的 Slf4jReporter 的 API，将度量输出到实现了 Slf4j 规

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

范的日志输出。

- GraphiteSink：借助 Metrics 提供的 GraphiteReporter 的 API，将度量输出到 Graphite（一个由 Python 实现的 Web 应用，采用 django 框架，用来收集服务器状态的监控系统）。

了解了 Sink 的类继承体系，我们挑选 Slf4jSink 作为 Spark 中 Sink 实现类的例子，来了解 Sink 具体该如何实现。Slf4jSink 的实现如代码清单 3-54 所示。

代码清单3-54　Slf4jSink的实现

```
private[spark] class Slf4jSink(
  val property: Properties,
  val registry: MetricRegistry,
  securityMgr: SecurityManager)
extends Sink {
val SLF4J_DEFAULT_PERIOD = 10
val SLF4J_DEFAULT_UNIT = "SECONDS"

val SLF4J_KEY_PERIOD = "period"
val SLF4J_KEY_UNIT = "unit"

val pollPeriod = Option(property.getProperty(SLF4J_KEY_PERIOD)) match {
  case Some(s) => s.toInt
  case None => SLF4J_DEFAULT_PERIOD
}

val pollUnit: TimeUnit = Option(property.getProperty(SLF4J_KEY_UNIT)) match {
  case Some(s) => TimeUnit.valueOf(s.toUpperCase())
  case None => TimeUnit.valueOf(SLF4J_DEFAULT_UNIT)
}

MetricsSystem.checkMinimalPollingPeriod(pollUnit, pollPeriod)

val reporter: Slf4jReporter = Slf4jReporter.forRegistry(registry)
  .convertDurationsTo(TimeUnit.MILLISECONDS)
  .convertRatesTo(TimeUnit.SECONDS)
  .build()

override def start() {
  reporter.start(pollPeriod, pollUnit)
}

override def stop() {
  reporter.stop()
}

override def report() {
  reporter.report()
 }
}
```



書籍是人类进步的阶梯

仅供非商业用途或交流学习使用

从 Slf4jSink 的实现可以看到，Slf4jSink 的 start、stop 及 report 实际都是代理了 Metrics 库中的 Slf4jReporter 的 start、stop 及 report 方法。Slf4jReporter 的 start 方法实际是其父类 ScheduledReporter 的 start 实现。而传递的两个参数 pollPeriod 和 pollUnit，正是被 Scheduled Reporter 使用作为定时器获取数据的周期和时间单位。有关 ScheduledReporter 中 start、stop 及 Slf4jReporter 的 report 方法的实现可以参阅附录 D。

3.5 小结

本章首先对 Spark 配置类 SparkConf 进行了介绍，SparkConf 是 Spark 中使用频度较高的类之一，Spark 处处都能看到它的身影。Spark 系统中很多不同的功能模块都需要使用 RPC 的功能，通过对 Netty 的 API 进行封装，提供了 Spark 内部统一的 RPC 框架。Spark 中有多个模块都采用了监听器模式实现事件与监听器，通过事件总线的抽象，非常有利于 Spark 的并发性能和功能扩展，其高度抽象对开发人员来讲，非常值得借鉴。度量系统采用了 Metrics 库，Spark 又抽象出了 Source、Sink 和 Instance，Source 和 Sink 的具体实现都依赖于 Metrics 的 API。



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

第 4 章 Chapter 4

SparkContext 的初始化

“合抱之木，生于毫末；九层之台，起于垒土；千里之行，始于足下。”

——《道德经》



本章导读

两千多年前，道家就已经总结出万事万物都是从最基本或最简单的工作开始的。参天大树最初是从一棵树芽生长而来的；摩天大厦是由一砖一瓦逐步建造而成的；如果你想游历祖国的大好河山，那么就得像徐霞客一样从迈出第一步开始。无论 Spark 提供的功能多么丰富，如果想要使用它，那么第一步就是对 SparkContext 进行初始化。对 SparkContext 进行初始化是你使用 Spark 所需要迈出的第一步，这一步看起来是多么的微不足道。

SparkContext 的初始化过程实际也是对 Driver 的初始化，这一准备工作是 Spark 应用程序提交与执行的前提。SparkContext 的初始化过程囊括了内部各个组件的初始化与准备，这些组件具有不同的功能，它们涉及网络通信、分布式、消息、存储、计算、缓存、度量、清理、文件服务、Web UI 的方方面面。这一切初始化工作都由 SparkContext 负责，所以我们不用过于担心它的内部复杂性。本章在介绍 SparkContext 初始化过程的同时，将向读者介绍各个组件的作用。有些组件只会作简单的介绍，后面的章节会安排详细的分析。

本章主要讲解的内容如下。

- SparkContext 概述。
- 创建 Spark 环境。
- Spark UI。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

- 其他组件的创建和启动。
- 环境更新。

4.1 SparkContext 概述

Spark 应用程序的提交离不开 Spark Driver，后者是驱动应用程序在 Spark 集群上执行的原动力。了解 Spark Driver 的初始化，有助于读者理解 Spark 应用程序与 Spark Driver 的关系。

Spark Driver 的初始化始终围绕着 SparkContext 的初始化。SparkContext 可以算得上是 Spark 应用程序的发动机引擎，轿车要想跑起来，首先要启动发动机。SparkContext 初始化完毕，才能向 Spark 集群提交应用程序。发动机只需以较低的转速，就可以在平坦的公路上游刃有余；在山区，你可能需要一台能够提供大功率的发动机，才能满足你转山的体验。发动机的参数都是通过驾驶员操作油门、档位等传送给发动机的，而 SparkContext 的配置参数则由 SparkConf 负责，SparkConf 就是你的操作面板。

SparkContext 是 Spark 中的元老级 API，从 0.x.x 版本就已经存在。有过 Spark 使用经验的部分读者也许感觉 SparkContext 已经太老了，然而 SparkContext 始终跟随着 Spark 的迭代不断向前。SparkContext 的内部“血液”也发生了很多翻天覆地的变化，有些内部组件废弃了，有些内部组件有了一些优化，而且还会不断地输入一些新鲜的“血液”。希望刚才这些描述没有吓到 Spark 的老用户，因为 Spark 的灵魂——Spark 核心原理，依然是那么令人熟悉。



在讲解 SparkContext 的初始化过程之前，我们先来认识下 SparkContext 中的各个组成部分，如图 4-1 所示。

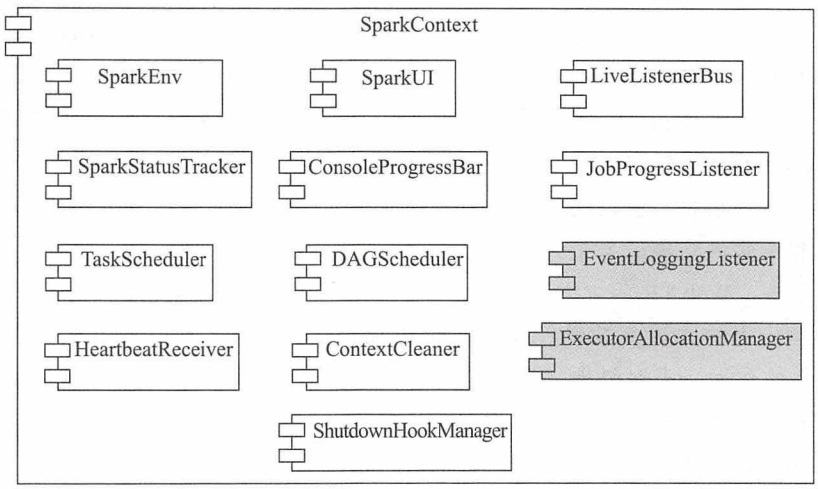


图 4-1 SparkContext 的组成[⊖]

⊖ 图中深色的组件是 SparkContext 中的可选组件。

仅供非商业用途或交流学习使用

从图 4-1 可以知道，SparkContext 主要由以下部分组成。

- SparkEnv：Spark 运行时环境。Executor 是处理任务的执行器，它依赖于 SparkEnv 提供的运行时环境。此外，在 Driver 中也包含了 SparkEnv，这是为了保证 local 模式下任务的执行。SparkEnv 内部包含了很多组件，例如，serializerManager、RpcEnv、BlockManager、mapOutputTracker 等，在这里只需要认识它们即可，第 5 章将会对这些组件进行具体介绍。
- LiveListenerBus：SparkContext 中的事件总线，可以接收各个使用方的事件，并且通过异步方式对事件进行匹配后调用 SparkListener 的不同方法。LiveListenerBus 的具体内容已经在 3.3 节详细介绍，此处不再赘述。
- SparkUI：Spark 的用户界面。SparkUI 间接依赖于计算引擎、调度系统、存储体系，作业（Job）、阶段（Stage）、存储、执行器（Executor）等组件的监控数据都会以 SparkListenerEvent 的形式投递到 LiveListenerBus 中，SparkUI 将从各个 SparkListener 中读取数据并显示到 Web 界面。
- SparkStatusTracker：提供对作业、Stage（阶段）等的监控信息。SparkStatusTracker 是一个低级的 API，这意味着只能提供非常脆弱的一致性机制。
- ConsoleProgressBar：利用 SparkStatusTracker 的 API，在控制台展示 Stage 的进度。由于 SparkStatusTracker 存在的一致性问题，所以 ConsoleProgressBar 在控制台的显示往往有一定的时延。
- DAGScheduler：DAG 调度器，是调度系统中的重要组件之一，负责创建 Job，将 DAG 中的 RDD 划分到不同的 Stage、提交 Stage 等。SparkUI 中有关 Job 和 Stage 的监控数据都来自 DAGScheduler。
- TaskScheduler：任务调度器，是调度系统中的重要组件之一。TaskScheduler 按照调度算法对集群管理器已经分配给应用程序的资源进行二次调度后分配给任务。TaskScheduler 调度的 Task 是由 DAGScheduler 创建的，所以 DAGScheduler 是 TaskScheduler 的前置调度。
- HeartbeatReceiver：心跳接收器。所有 Executor 都会向 HeartbeatReceiver 发送心跳信息，HeartbeatReceiver 接收到 Executor 的心跳信息后，首先更新 Executor 的最后可见时间，然后将此信息交给 TaskScheduler 作进一步处理。
- ContextCleaner：上下文清理器。ContextCleaner 实际用异步方式清理那些超出应用作用域范围的 RDD、ShuffleDependency 和 Broadcast 等信息。
- JobProgressListener：作业进度监听器。JobProgressListener 在 3.3.3 节介绍 SparkListener 的继承体系时提到过，根据之前的介绍我们知道，JobProgressListener 将注册到 LiveListenerBus 中作为事件监听器之一使用。
- EventLoggingListener：将事件持久化到存储的监听器，是 SparkContext 中的可选组件。当 spark.eventLog.enabled 属性为 true 时启用。



仅供非商业用途或交流学习使用

- ExecutorAllocationManager：Executor 动态分配管理器。顾名思义，可以根据工作负载动态调整 Executor 的数量。在配置 spark.dynamicAllocation.enabled 属性为 true 的前提下，在非 local 模式下或者当 spark.dynamicAllocation.testing 属性为 true 时启用。
- ShutdownHookManager：用于设置关闭钩子的管理器。可以给应用设置关闭钩子，这样就可以在 JVM 进程退出时，执行一些清理工作。
- 除了以上介绍的这些 SparkContext 内部组件，SparkContext 内部还包括以下属性。
- creationSite[⊖]：类型为 CallSite，其中保存着线程栈中最靠近栈顶的用户定义的类及最靠近栈底的 Scala 或者 Spark 核心类信息，CallSite 的 shortForm 属性保存着以上信息的简短描述，CallSite 的 longForm 属性则保存着以上信息的完整描述。Spark 自带的 examples 项目中有对单词进行计数的应用例子 JavaWordCount，运行 JavaWordCount 得到的 CallSite 对象的属性值分别如下。
  - shortForm：getOrCreate at JavaWordCount.java:48。
  - longForm：org.apache.spark.sql.SparkSession$Builder.getOrCreate(SparkSession.scala:860) org.apache.spark.examples.JavaWordCount.main(JavaWordCount.java:48)。
- allowMultipleContexts：是否允许多个 SparkContext 实例。默认为 false，可以通过属性 spark.driver.allowMultipleContexts 来控制。
- startTime：SparkContext 启动的时间戳。
- stopped：标记 SparkContext 是否已经停止的状态，采用原子类型 AtomicBoolean。
- addedFiles：用于每个本地文件的 URL 与添加此文件到 addedFiles 时的时间戳之间的映射缓存。
- addedJars：用于每个本地 Jar 文件的 URL 与添加此文件到 addedJars 时的时间戳之间的映射缓存。
- persistentRdds：用于对所有持久化的 RDD 保持跟踪。
- executorEnvs：用于存储环境变量。executorEnvs 中环境变量都将传递给执行任务的 Executor 使用。
- sparkUser：当前系统的登录用户，也可以通过系统环境变量 SPARK_USER 进行设置。这里使用的 Utils 的 getCurrentUserName 方法的更多介绍，请阅读附录 A。
- checkpointDir：RDD 计算过程中保存检查点时所需要的目录。
- localProperties：由 InheritableThreadLocal 保护的线程本地变量，其中的属性值可以沿着线程栈传递下去，供用户使用。
- _conf：SparkContext 的配置，通过调用 SparkConf 的 clone 方法的克隆体。在 SparkContext 初始化的过程中会对 conf 中的配置信息做校验，例如，用户必须给自己



⊖ creationSite 通过调用 Utils.getCallSite 获得，Utils.getCallSite 的详细信息见附录 A。

仅供非商业用途或交流学习使用

的应用程序设置 spark.master（采用的部署模式）和 spark.app.name（用户应用的名称）；用户设置的 spark.master 属性为 yarn 时，spark.submit.deployMode 属性必须为 cluster，且必须设置 spark.yarn.app.id 属性。

- _jars：用户设置的 Jar 文件。当用户选择的部署模式是 YARN 时，_jars 是由 spark.jars 属性指定的 Jar 文件和 spark.yarn.dist.jars 属性指定的 Jar 文件的并集。其他模式下只采用由 spark.jars 属性指定的 Jar 文件。这里使用了 Utils 的 getUserJars 方法，其具体介绍请阅读附录 A。
- _files：用户设置的文件。可以使用 spark.files 属性进行指定。
- _eventLogDir：事件日志的路径。当 spark.eventLog.enabled 属性为 true 时启用。默认为 /tmp/spark-events，也可以通过 spark.eventLog.dir 属性指定。
- _eventLogCodec：事件日志的压缩算法。当 spark.eventLog.enabled 属性与 spark.eventLog.compress 属性皆为 true 时启用。压缩算法默认为 lz4，也可以通过 spark.io.compression.codec 属性指定。Spark 目前支持的压缩算法包括 lzf、snappy 和 lz4 这 3 种。
- _hadoopConfiguration：Hadoop 的配置信息，具体根据 Hadoop（Hadoop 2.0 之前的版本）和 Hadoop YARN（Hadoop2.0+ 的版本）的环境有所区别。如果系统属性 SPARK_YARN_MODE 为 true 或者环境变量 SPARK_YARN_MODE 为 true，那么将会是 YARN 的配置，否则为 Hadoop 配置。
- _executorMemory：Executor 的内存大小。默认值为 1024MB。可以通过设置环境变量（SPARK_MEM 或者 SPARK_EXECUTOR_MEMORY）或者 spark.executor.memory 属性指定。其中，spark.executor.memory 的优先级最高，SPARK_EXECUTOR_MEMORY 次之，SPARK_MEM 是老版本 Spark 遗留下来的配置方式，未来将会废弃。
- _applicationId：当前应用的标识。TaskScheduler 启动后会创建应用标识，SparkContext 中的 _applicationId 就是通过调用 TaskScheduler 的 applicationId 方法获得的。
- _applicationAttemptId：当前应用尝试执行的标识。Spark Driver 在执行时会多次尝试执行，每次尝试都将生成一个标识来代表应用尝试执行的身份。
- _listenerBusStarted：LiveListenerBus 是否已经启动的标记。
- nextShuffleId：类型为 AtomicInteger，用于生成下一个 Shuffle 的身份标识。
- nextRddId：类型为 AtomicInteger，用于生成下一个 RDD 的身份标识。

4.2　创建 Spark 环境

在 Spark 中，凡是需要执行任务的地方就需要 SparkEnv。在生产环境中，SparkEnv 往

仅供非商业用途或交流学习使用

往运行于不同节点的 Executor 中。但是由于 local 模式在本地执行的需要，因此在 Driver 本地的 Executor 也需要 SparkEnv。SparkContext 中创建 SparkEnv 的实现如代码清单 4-1 所示。

代码清单4-1 创建SparkEnv

```
private[spark] val listenerBus = new LiveListenerBus(this)
  _jobProgressListener = new JobProgressListener(_conf)
  listenerBus.addListener(jobProgressListener)
  _env = createSparkEnv(_conf, isLocal, listenerBus)
  SparkEnv.set(_env)
```

因为 SparkEnv 内的很多组件都将向 LiveListenerBus 的事件队列中投递事件，所以在代码清单 4-1 中首先创建 LiveListenerBus 和 JobProgressListener，然后将 JobProgressListener 添加到 LiveListenerBus 的监听器列表中，最后将 LiveListenerBus 通过 SparkEnv 的构造器传递给 SparkEnv 及 SparkEnv 内部的组件。JobProgressListener 继承自 SparkListener，LiveListenerBus 和 SparkListener 的详细内容已在 3.3 节介绍，此处不再赘述。

createDriverEnv 方法用于创建 SparkEnv，根据 createDriverEnv 这个方法名，我们知道此方法将为 Driver 实例创建 SparkEnv。调用 createSparkEnv 方法创建完 SparkEnv 后，SparkEnv 实例的引用将通过 SparkEnv 的 set 方法设置到 SparkEnv 伴生对象⊖的 env 属性中，类似于设置为 Java 类的静态属性，这将便于在任何需要 SparkEnv 的地方，通过伴生对象的 get 方法获取 SparkEnv。

createSparkEnv 方法创建 SparkEnv 的代码如下。

```
private[spark] def createSparkEnv(
    conf: SparkConf,
    isLocal: Boolean,
    listenerBus: LiveListenerBus): SparkEnv = {
  SparkEnv.createDriverEnv(conf, isLocal, listenerBus, SparkContext.
      numDriverCores(master))
}
```

可以看到实际调用了 SparkEnv 的 createDriverEnv 方法来创建 SparkEnv。SparkEnv 的 createDriverEnv 方法的实现如下。

```
private[spark] def createDriverEnv(
    conf: SparkConf,
    isLocal: Boolean,
    listenerBus: LiveListenerBus,
    numCores: Int,
    mockOutputCommitCoordinator: Option[OutputCommitCoordinator] = None):
        SparkEnv = {
  assert(conf.contains(DRIVER_HOST_ADDRESS),
    s"${DRIVER_HOST_ADDRESS.key} is not set on the driver!")
```

⊖ 伴生对象是 Scala 语言的语法特性。Scala 中的伴生对象可以提供类似于 Java 中的静态方法、静态属性等的定义，也可以支持类似于 Java 中的 main 方法。更多内容请读者阅读 Scala 语言相关资料。

仅供非商业用途或交流学习使用


```
  assert(conf.contains("spark.driver.port"), "spark.driver.port is not set on the
driver!")
  val bindAddress = conf.get(DRIVER_BIND_ADDRESS)
  val advertiseAddress = conf.get(DRIVER_HOST_ADDRESS)
  val port = conf.get("spark.driver.port").toInt
  val ioEncryptionKey = if (conf.get(IO_ENCRYPTION_ENABLED)) {
    Some(CryptoStreamUtils.createKey(conf))
  } else {
    None
  }
  create(
    conf,
    SparkContext.DRIVER_IDENTIFIER,
    bindAddress,
    advertiseAddress,
    port,
    isLocal,
    numCores,
    ioEncryptionKey,
    listenerBus = listenerBus,
    mockOutputCommitCoordinator = mockOutputCommitCoordinator
  )
}
```

createDriverEnv 方法首先从 SparkConf 中获取 4 个属性。

- bindAddress：Driver 实例的 host。此属性通过从 SparkConf 中获取 DRIVER_BIND_ADDRESS 指定的属性值。DRIVER_BIND_ADDRESS 的定义如下。



```
private[spark] val DRIVER_HOST_ADDRESS = ConfigBuilder("spark.driver.host")
  .doc("Address of driver endpoints.")
  .stringConf
  .createWithDefault(Utils.localHostName())

private[spark] val DRIVER_BIND_ADDRESS = ConfigBuilder("spark.driver.bind
    Address")
  .doc("Address where to bind network listen sockets on the driver.")
  .fallbackConf(DRIVER_HOST_ADDRESS)
```

根据 DRIVER_BIND_ADDRESS 的定义，说明按照优先级从高到低，可以通过 spark.driver.bindAddress 属性、spark.driver.host 属性及调用 Utils 的 localHostName 方法获得 bindAddress。Utils 的 localHostName 方法的实现请参阅附录 A。

- advertiseAddress：Driver 实例对外宣称的 host。可以通过 spark.driver.host 属性或者 Utils 的 localHostName 方法获得。
- port：Driver 实例的端口，可以通过 spark.driver.port 属性指定。
- ioEncryptionKey：I/O 加密的密钥。当 spark.io.encryption.enabled 属性为 true 时，调用 CryptoStreamUtils 的 createKey 方法创建密钥。

真正创建 SparkEnv 的实现都在 create 方法中，由于 SparkEnv 是 Driver 和 Executor 实例中都有的组件，本书将在第 5 章对 SparkEnv 作详细介绍。

仅供非商业用途或交流学习使用

4.3 SparkUI 的实现

任何系统都需要提供监控功能，否则在运行期间发生一些异常时，我们将会束手无策。也许有人说，可以增加日志来解决这个问题。日志只能解决你的程序逻辑在运行期的监控，进而发现 Bug，以及提供对业务有帮助的调试信息，当你的 JVM 进程崩溃或者程序响应速度很慢时，这些日志将毫无用处。好在 JVM 提供了 jstat、jstack、jinfo、jmap、jhat 等工具帮助我们分析，更有 VisualVM 的可视化界面以更加直观的方式对 JVM 运行期的状况进行监控。此外，像 Tomcat、Hadoop 等服务都提供了基于 Web 的监控页面，用浏览器能访问具有样式及布局，并提供丰富监控数据的页面无疑是一种简单、高效的方式。

Spark 自然也提供了 Web 页面来浏览监控数据，而且 Master、Worker、Driver 根据自身功能提供了不同内容的 Web 监控页面。无论是 Master、Worker，还是 Driver，它们都使用了统一的 Web 框架 WebUI。Master、Worker 及 Driver 分别使用 MasterWebUI、WorkerWebUI 及 SparkUI 提供的 Web 界面服务，后三者都继承自 WebUI，并增加了个性化的功能。此外，在 YARN 或 Mesos 模式下还有 WebUI 的另一个扩展实现 HistoryServer。HistoryServer 将会展现已经运行完成的应用程序信息。本章以 SparkUI 为例，并深入分析 WebUI 的框架体系。

4.3.1 SparkUI 概述



在大型分布式系统中，采用事件监听机制是最常见的。为什么要使用事件监听机制？假如 SparkUI 采用 Scala 的函数调用方式，那么随着整个集群规模的增加，对函数的调用会越来越多，最终会受到 Driver 所在 JVM 的线程数量限制而影响监控数据的更新，甚至出现监控数据无法及时显示给用户的情况。由于函数调用多数情况下是同步调用，这就导致线程被阻塞，在分布式环境中，还可能因为网络问题，导致线程被长时间占用。将函数调用更换为发送事件，事件的处理是异步的，当前线程可以继续执行后续逻辑进而被快速释放。线程池中的线程还可以被重用，这样整个系统的并发度会大大增加。发送的事件会存入缓存，由定时调度器取出后，分配给监听此事件的监听器对监控数据进行更新。SparkUI 就是这样的服务，它的构成如图 4-2 所示。

图 4-2 展示了 SparkUI 中的各个组件，这里对这些组件作简单介绍。

1）SparkListenerEvent 事件的来源：包括 DAGScheduler、SparkContext、DriverEndpoint、BlockManagerMasterEndpoint 及 LocalSchedulerBackend 等，这些组件将会产生各种 SparkListenerEvent，并发送到 listenerBus 的事件队列中。DriverEndpoint 是 Driver 在 Standalone 或 local-cluster 模式下与其他组件进行通信的组件，本书将在 9.9.2 节详细介绍。BlockManagerMasterEndpoint 是 Driver 对分配给应用的所有 Executor 及其 BlockManager 进行统一管理的组件，本书将在 6.8 节详细介绍。LocalSchedulerBackend 是 local 模式下的调度后端接口，用于给任务分配资源或对任务的状态进行更新，本书将在 7.8.2 节详细

仅供非商业用途或交流学习使用

介绍。

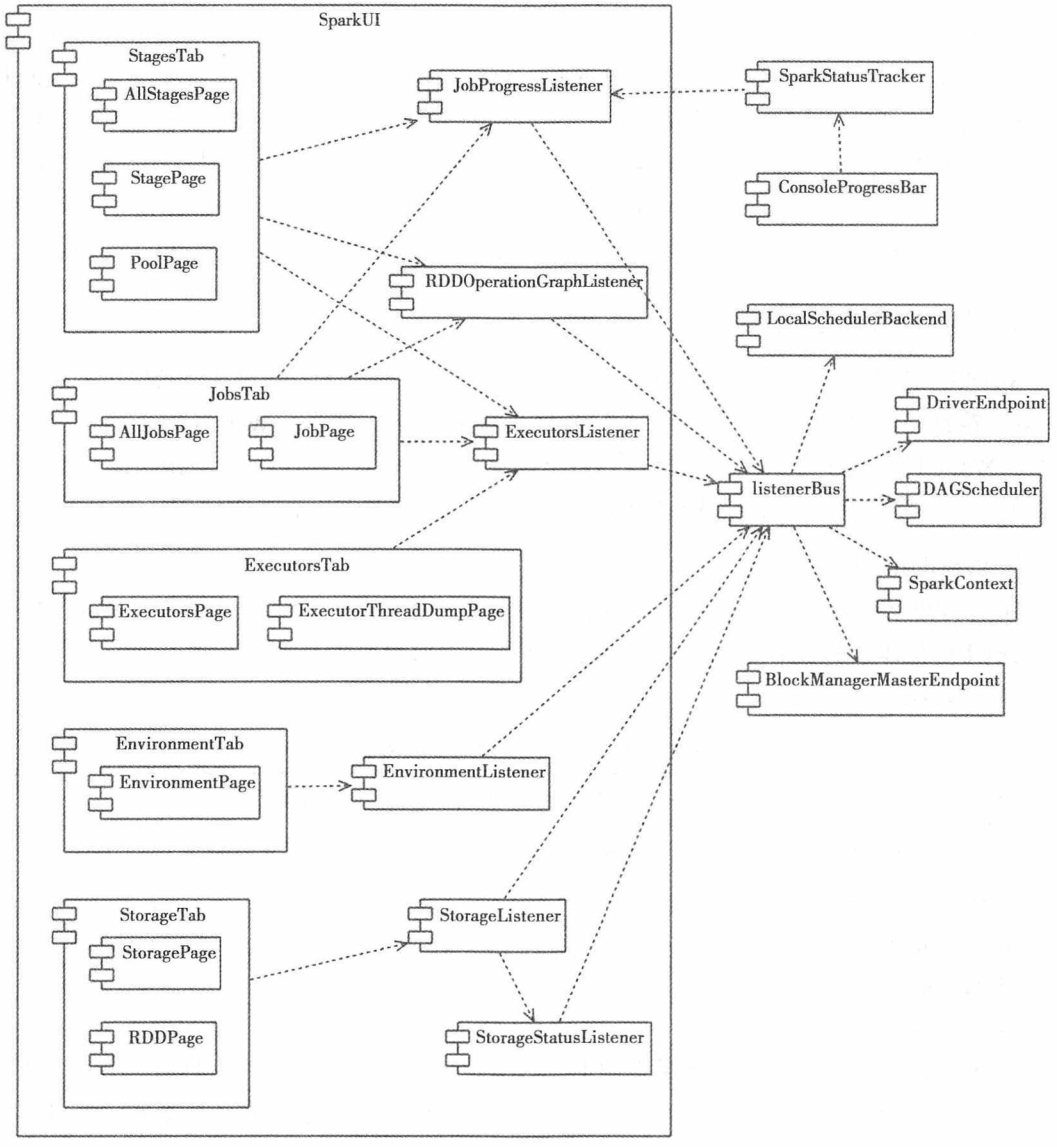


图 4-2　SparkUI 的组成

2）事件总线 listenerBus。根据 3.3 节对事件总线的介绍，我们知道 listenerBus 通过定时器将 SparkListenerEvent 事件匹配到具体的 SparkListener，进而改变各个 SparkListener 中的统计监控数据。

3）SparkUI 的界面。各个 SparkListener 内的统计监控数据将会被各种标签页和具体页面展示到 Web 界面。标签页有 StagesTab、JobsTab、ExecutorsTab、EnvironmentTab 及 Storage

仅供非商业用途或交流学习使用

Tab。每个标签页中包含若干个页面，例如，StagesTab 标签页中包含了 AllStagesPage、StagePage 及 PoolPage 三个页面。

4）控制台的展示。细心的读者会发现图 4-2 中还有 SparkStatusTracker（Spark 状态跟踪器）和 ConsoleProgressBar（控制台进度条）两个组件。SparkStatusTracker 负责对 Job 和 Stage 的监控，实际也是使用了 JobProgressListener 中的监控数据，并额外进行了一些加工。ConsoleProgressBar 负责将 SparkStatusTracker 提供的数据打印到控制台上。从最终展现的角度来看，SparkStatusTracker 和 ConsoleProgressBar 不应该属于 SparkUI 的组成部分，但是由于其实现与 JobProgressListener 密切相关，所以将它们也放在了 SparkUI 的内容中。

4.3.2 WebUI 框架体系

SparkUI 构建在 WebUI 的框架体系之上，因此应当首先了解 WebUI。WebUI 定义了一种 Web 界面展现的框架，并提供返回 Json 格式数据的 Web 服务。WebUI 用于展示一组标签页，WebUITab 定义了标签页的规范。每个标签页中包含着一组页面，WebUIPage 定义了页面的规范。我们将首先了解 WebUIPage 和 WebUITab，最后从整体来看 WebUI。

1. WebUIPage 的定义

任何 Web 界面往往由多个页面组成，每个页面都将提供不同的内容展示。WebUIPage 是 WebUI 框架体系的页节点，定义了所有页面应当遵循的规范。抽象类 WebUIPage 的定义如代码清单 4-2 所示。



代码清单4-2 WebUIPage的定义

```
private[spark] abstract class WebUIPage(var prefix: String) {
  def render(request: HttpServletRequest): Seq[Node]
  def renderJson(request: HttpServletRequest): JValue = JNothing
}
```

WebUIPage 定义了两个方法。

- render：渲染页面。
- renderJson：生成 JSON。

WebUIPage 在 WebUI 框架体系中的上一级节点（也可以称为父亲）可以是 WebUI 或者 WebUITab，其成员属性 prefix 将与上级节点的路径一起构成当前 WebUIPage 的访问路径。

2. WebUITab 的定义

有时候 Web 界面需要将多个页面作为一组内容放置在一起，这时候标签页是常见的展现形式。标签页 WebUITab 定义了所有标签页的规范，并用于展现一组 WebUIPage。抽象类 WebUITab 的定义如代码清单 4-3 所示。

代码清单4-3 WebUITab的定义

```
private[spark] abstract class WebUITab(parent: WebUI, val prefix: String) {
```

仅供非商业用途或交流学习使用


```
  val pages = ArrayBuffer[WebUIPage]()
  val name = prefix.capitalize
  def attachPage(page: WebUIPage) {
    page.prefix = (prefix + "/" + page.prefix).stripSuffix("/")
    pages += page
  }
  def headerTabs: Seq[WebUITab] = parent.getTabs

  def basePath: String = parent.getBasePath
}
```

根据代码清单 4-3，可以看到 WebUITab 有 4 个成员属性。

- parent：上一级节点，即父亲。WebUITab 的父亲只能是 WebUI。
- prefix：当前 WebUITab 的前缀。prefix 将与上级节点的路径一起构成当前 WebUITab 的访问路径。
- pages：当前 WebUITab 所包含的 WebUIPage 的缓冲数组。
- name：当前 WebUITab 的名称。name 实际是将 prefix 的首字母转换成大写字母后取得。

此外，WebUITab 还有 3 个成员方法，下面介绍它们的作用。

- attachPage：首先将当前 WebUITab 的前缀与 WebUIPage 的前缀拼接，作为 WebUIPage 的访问路径，然后向 pages 中添加 WebUIPage。
- headerTabs：获取父亲 WebUI 中的所有 WebUITab。此方法实际通过调用父亲 WebUI 的 getTabs 方法实现，getTabs 方法请参阅下面“WebUI 的定义”部分。
- basePath：获取父亲 WebUI 的基本路径。此方法实际通过调用父亲 WebUI 的 getBasePath 方法实现，getBasePath 方法请参阅下面“WebUI 的定义”部分。

3. WebUI 的定义

WebUI 是 Spark 实现的用于提供 Web 界面展现的框架，凡是需要页面展现的地方都可以继承它来完成。WebUI 定义了 WebUI 框架体系的规范。为便于理解，首先明确 WebUI 中各个成员属性的含义。

- securityManager：SparkEnv 中创建的安全管理器 SecurityManager，5.2 节对 SecurityManager 有详细介绍。
- sslOptions：使用 SecurityManager 获取 spark.ssl.ui 属性指定的 WebUI 的 SSL(Secure Sockets Layer，安全套接层）选项。
- port：WebUI 对外服务的端口，可以使用 spark.ui.port 属性进行配置。
- conf：即 SparkConf。
- basePath：WebUI 的基本路径，basePath 默认为空字符串。
- name：WebUI 的名称，Spark UI 的 name 为 SparkUI。
- tabs：WebUITab 的缓冲数组。

仅供非商业用途或交流学习使用

- handlers：ServletContextHandler 的缓冲数组。ServletContextHandler 是 Jetty 提供的 API，负责对 ServletContext 进行处理。ServletContextHandler 的使用及 Jetty 的更多内容可以参阅附录 C。
- pageToHandlers：WebUIPage 与 ServletContextHandler 缓冲数组之间的映射关系。由于 WebUIPage 的两个方法 render 和 renderJson 分别需要由一个对应的 ServletContextHandler 处理，所以一个 WebUIPage 对应两个 ServletContextHandler。
- serverInfo：用于缓存 ServerInfo，即 WebUI 的 Jetty 服务器信息。
- publicHostName：当前 WebUI 的 Jetty 服务的主机名。优先采用系统环境变量 SPARK_PUBLIC_DNS 指定的主机名，否则采用 spark.driver.host 属性指定的 host，在没有前两个配置的时候，将默认使用工具类 Utils 的 localHostName 方法（详见附录 A）返回的主机名。
- className：过滤了 $ 符号的当前类的简单名称。className 是通过 Utils 的 getFormattedClassName 方法得到的。getFormattedClassName 方法的实现请参看附录 A。

了解了 WebUI 的成员属性，现在就可以理解其提供的各个方法了。WebUI 提供的方法有以下几种。

- getBasePath：获取 basePath。
- getTabs：获取 tabs 中的所有 WebUITab，并以 Scala 的序列返回。
- getHandlers：获取 handlers 中的所有 ServletContextHandler，并以 Scala 的序列返回。
- getSecurityManager：获取 securityManager。
- attachHandler：给 handlers 缓存数组中添加 ServletContextHandler，并且将此 ServletContextHandler 通过 ServerInfo 的 addHandler 方法添加到 Jetty 服务器中。attachHandler 的实现如代码清单 4-4 所示。ServerInfo 的 addHandler 方法请参阅附录 C。



代码清单4-4 attachHandler的实现

```
def attachHandler(handler: ServletContextHandler) {
  handlers += handler
  serverInfo.foreach(_.addHandler(handler))
}
```

- detachHandler：从 handlers 缓存数组中移除 ServletContextHandler，并且将此 ServletContextHandler 通过 ServerInfo 的 removeHandler 方法从 Jetty 服务器中移除。detachHandler 的实现如代码清单 4-5 所示。ServerInfo 的 removeHandler 方法请参阅附录 C。

代码清单4-5 detachHandler的实现

```
def detachHandler(handler: ServletContextHandler) {
```

仅供非商业用途或交流学习使用


```
    handlers -= handler
    serverInfo.foreach(_.removeHandler(handler))
  }
```

- attachPage：首先调用工具类 JettyUtils 的 createServletHandler 方法给 WebUIPage 创建与 render 和 renderJson 两个方法分别关联的 ServletContextHandler，然后通过 attachHandler 方法添加到 handlers 缓存数组与 Jetty 服务器中，最后把 WebUIPage 与这两个 ServletContextHandler 的映射关系更新到 pageToHandlers 中。attachPage 的实现如代码清单 4-6 所示。

代码清单4-6　attachPage的实现

```
def attachPage(page: WebUIPage) {
  val pagePath = "/" + page.prefix
  val renderHandler = createServletHandler(pagePath,
    (request: HttpServletRequest) => page.render(request), securityManager, conf,
        basePath)
  val renderJsonHandler = createServletHandler(pagePath.stripSuffix("/") + "/
     json",
     (request: HttpServletRequest) => page.renderJson(request), securityManager,
conf, basePath)
  attachHandler(renderHandler)
  attachHandler(renderJsonHandler)
  val handlers = pageToHandlers.getOrElseUpdate(page, ArrayBuffer[ServletContext
     Handler]())
  handlers += renderHandler
}
```



- detachPage：作用与 attachPage 相反。detachPage 的实现如代码清单 4-7 所示。

代码清单4-7　detachPage的实现

```
def detachPage(page: WebUIPage) {
  pageToHandlers.remove(page).foreach(_.foreach(detachHandler))
}
```

- attachTab：首先向 tabs 中添加 WebUITab，然后给 WebUITab 中的每个 WebUIPage 施加 attachPage 方法。attachTab 的实现如代码清单 4-8 所示。

代码清单4-8　attachTab的实现

```
def attachTab(tab: WebUITab) {
  tab.pages.foreach(attachPage)
  tabs += tab
}
```

- detachTab：作用与 attachTab 相反。detachTab 的实现如代码清单 4-9 所示。

代码清单4-9　detachTab的实现

```
def detachTab(tab: WebUITab) {
  tab.pages.foreach(detachPage)
```

仅供非商业用途或交流学习使用


```
    tabs -= tab
  }
```

- addStaticHandler：首先调用工具类 JettyUtils 的 createStaticHandler 方法创建静态文件服务的 ServletContextHandler，然后施加 attachHandler 方法。addStaticHandler 的实现如代码清单 4-10 所示。JettyUtils 的 createStaticHandler 方法的实现见附录 C。

代码清单4-10 addStaticHandler的实现

```
def addStaticHandler(resourceBase: String, path: String): Unit = {
  attachHandler(JettyUtils.createStaticHandler(resourceBase, path))
}
```

- removeStaticHandler：作用与 addStaticHandler 相反。removeStaticHandler 的实现如代码清单 4-11 所示。

代码清单4-11 removeStaticHandler的实现

```
def removeStaticHandler(path: String): Unit = {
  handlers.find(_.getContextPath() == path).foreach(detachHandler)
}
```

- initialize：用于初始化 WebUI 服务中的所有组件。WebUI 中此方法未实现，需要子类实现。
- bind：启动与 WebUI 绑定的 Jetty 服务。bind 方法的实现如代码清单 4-12 所示。

代码清单4-12 bind的实现

```
def bind() {
  assert(!serverInfo.isDefined, s"Attempted to bind $className more than once!")
  try {
    val host = Option(conf.getenv("SPARK_LOCAL_IP")).getOrElse("0.0.0.0")
    serverInfo = Some(startJettyServer(host, port, sslOptions, handlers, conf,
        name))
    logInfo(s"Bound $className to $host, and started at $webUrl")
  } catch {
    case e: Exception =>
      logError(s"Failed to bind $className", e)
      System.exit(1)
  }
}
```

- webUrl：获取 WebUI 的 Web 界面的 URL。webUrl 的实现如下。

```
def webUrl: String = shttp://$publicHostName:$boundPort
```

- boundPort：获取 WebUI 的 Jetty 服务的端口。boundPort 的实现如下。

```
def boundPort: Int = serverInfo.map(_.boundPort).getOrElse(-1)
```

- stop：停止 WebUI。实际是停止 WebUI 底层的 Jetty 服务。stop 方法的实现如代码清单 4-13 所示。

仅供非商业用途或交流学习使用

代码清单4-13　stop方法的实现

```
def stop() {
  assert(serverInfo.isDefined,
    s"Attempted to stop $className before binding to a server!")
  serverInfo.get.stop()
}
```

4.3.3　创建 SparkUI

在 SparkContext 的初始化过程中，会创建 SparkUI。有了对 WebUI 的总体认识，现在是时候了解 SparkContext 是如何构造 SparkUI 的了。SparkUI 是 WebUI 框架的使用范例，了解了 SparkUI 的创建过程，读者对 MasterWebUI、WorkerWebUI 及 HistoryServer 的创建过程也必然了然于心。创建 SparkUI 的代码如下。

```
_statusTracker = new SparkStatusTracker(this)

_progressBar =
  if (_conf.getBoolean("spark.ui.showConsoleProgress", true) && !log.isInfoEnabled) {
    Some(new ConsoleProgressBar(this))
  } else {
    None
  }

_ui =
  if (conf.getBoolean("spark.ui.enabled", true)) {
    Some(SparkUI.createLiveUI(this, _conf, listenerBus, _jobProgressListener,
      _env.securityManager, appName, startTime = startTime))
  } else {
    // For tests, do not enable the UI
    None
  }
_ui.foreach(_.bind())
```

这段代码的执行步骤如下。

1）创建 Spark 状态跟踪器 SparkStatusTracker。

2）创建 ConsoleProgressBar。可以配置 spark.ui.showConsoleProgress 属性为 false，取消对 ConsoleProgressBar 的创建，此属性默认为 true。

3）调用 SparkUI 的 createLiveUI 方法创建 SparkUI。

4）给 SparkUI 绑定端口。SparkUI 继承自 WebUI，因此调用了代码清单 4-12 中 WebUI 的 bind 方法启动 SparkUI 底层的 Jetty 服务。

上述步骤中，第 1）、2）、4）步都很简单，所以着重来分析第 3）步。SparkUI 的 createLiveUI 的实现如下。

```
def createLiveUI(
    sc: SparkContext,
    conf: SparkConf,
    listenerBus: SparkListenerBus,
```

仅供非商业用途或交流学习使用


```
    jobProgressListener: JobProgressListener,
    securityManager: SecurityManager,
    appName: String,
    startTime: Long): SparkUI = {
  create(Some(sc), conf, listenerBus, securityManager, appName,
    jobProgressListener = Some(jobProgressListener), startTime = startTime)
}
```

可以看到 SparkUI 的 createLiveUI 方法中调用了 create 方法。create 的实现如下。

```
private def create(
    sc: Option[SparkContext],
    conf: SparkConf,
    listenerBus: SparkListenerBus,
    securityManager: SecurityManager,
    appName: String,
    basePath: String = "",
    jobProgressListener: Option[JobProgressListener] = None,
    startTime: Long): SparkUI = {

  val _jobProgressListener: JobProgressListener = jobProgressListener.getOrElse {
    val listener = new JobProgressListener(conf)
    listenerBus.addListener(listener)
    listener
  }

  val environmentListener = new EnvironmentListener
  val storageStatusListener = new StorageStatusListener(conf)
  val executorsListener = new ExecutorsListener(storageStatusListener, conf)
  val storageListener = new StorageListener(storageStatusListener)
  val operationGraphListener = new RDDOperationGraphListener(conf)

  listenerBus.addListener(environmentListener)
  listenerBus.addListener(storageStatusListener)
  listenerBus.addListener(executorsListener)
  listenerBus.addListener(storageListener)
  listenerBus.addListener(operationGraphListener)

  new SparkUI(sc, conf, securityManager, environmentListener, storageStatusListener,
    executorsListener, _jobProgressListener, storageListener, operationGraphListener,
    appName, basePath, startTime)
}
```



可以看到，create 方法里除了 JobProgressListener 是外部传入的之外，又增加了一些 SparkListener，例如，用于对 JVM 参数、Spark 属性、Java 系统属性、classpath 等进行监控的 EnvironmentListener；用于维护 Executor 的存储状态的 StorageStatusListener；用于准备将 Executor 的信息展示在 ExecutorsTab 的 ExecutorsListener；用于准备将 Executor 相关存储信息展示在 BlockManagerUI 的 StorageListener；用于构建 RDD 的 DAG（有向无关图）的 RDDOperationGraphListener 等。根据代码清单 4-1，JobProgressListener 早已被添加到 listenerBus 的监听器列表中，所以此处只需要将另外 5 个 SparkListener 的实现添加到 listenerBus 的监听器列表中。最后使用 SparkUI 的构造器创建 SparkUI。

仅供非商业用途或交流学习使用

1. SparkUI 的初始化

调用 SparkUI 的构造器创建 SparkUI，实际也是对 SparkUI 的初始化过程。在介绍初始化之前，先来看看 SparkUI 中的两个成员属性。

- killEnabled：标记当前 SparkUI 能否提供“杀死”Stage 或者 Job 的链接。
- appId：当前应用的 ID。

SparkUI 的构造过程中会执行 initialize 方法，其实现如代码清单 4-14 所示。

代码清单4-14　SparkUI的初始化

```
def initialize() {
  val jobsTab = new JobsTab(this)
  attachTab(jobsTab)
  val stagesTab = new StagesTab(this)
  attachTab(stagesTab)
  attachTab(new StorageTab(this))
  attachTab(new EnvironmentTab(this))
  attachTab(new ExecutorsTab(this))
  attachHandler(createStaticHandler(SparkUI.STATIC_RESOURCE_DIR, "/static"))
  attachHandler(createRedirectHandler("/", "/jobs/", basePath = basePath))
  attachHandler(ApiRootResource.getServletHandler(this))
  // These should be POST only, but, the YARN AM proxy won't proxy POSTs
  attachHandler(createRedirectHandler(
    "/jobs/job/kill", "/jobs/", jobsTab.handleKillRequest, httpMethods = Set("GET",
        "POST")))
  attachHandler(createRedirectHandler(
    "/stages/stage/kill", "/stages/", stagesTab.handleKillRequest,
    httpMethods = Set("GET", "POST")))
}
initialize()
```



根据代码清单 4-14，SparkUI 的初始化步骤如下。

1）构建页面布局并给每个 WebUITab 中的所有 WebUIPage 创建对应的 ServletContextHandler。这一步使用了代码清单 4-8 中展示的 attachTab 方法。

2）调用 JettyUtils 的 createStaticHandler 方法创建对静态目录 org/apache/spark/ui/static 提供文件服务的 ServletContextHandler，并使用 attachHandler 方法追加到 SparkUI 的服务中。

3）调用 JettyUtils 的 createRedirectHandler 方法创建几个将用户对源路径的请求重定向到目标路径的 ServletContextHandler。例如，将用户对根路径“/”的请求重定向到目标路径“/jobs/”的 ServletContextHandler。

2. SparkUI 的页面布局与展示

SparkUI 究竟是如何实现页面布局及展示的？由于所有标签页都继承了 SparkUITab，所以我们先来看看 SparkUITab 的实现。

```
private[spark] abstract class SparkUITab(parent: SparkUI, prefix: String)
  extends WebUITab(parent, prefix) {
```

仅供非商业用途或交流学习使用


```
  def appName: String = parent.getAppName
}
```

根据上述代码，我们知道 SparkUITab 继承了 WebUITab，并在实现中增加了一个用于获取当前应用名称的方法 appName。EnvironmentTab 是用于展示 JVM、Spark 属性、系统属性、类路径等相关信息的标签页，由于其实现简单且能说明问题，所以本节挑选 EnvironmentTab 作为示例解答本节一开始提出的问题。

EnvironmentTab 的实现如代码清单 4-15 所示。

代码清单4-15 EnvironmentTab的实现

```
private[ui] class EnvironmentTab(parent: SparkUI) extends SparkUITab(parent,
    "environment") {
  val listener = parent.environmentListener
  attachPage(new EnvironmentPage(this))
}
```

根据代码清单 4-15，我们知道 EnvironmentTab 引用了 SparkUI 的 environmentListener（类型为 EnvironmentListener），并且包含 EnvironmentPage 这个页面。EnvironmentTab 通过调用 attachPage 方法将 EnvironmentPage 与 Jetty 服务关联起来。根据代码清单 4-6 中 attachPage 的实现，创建的 renderHandler 将采用偏函数 (request: HttpServletRequest) => page.render(request) 处理请求，因而会调用 EnvironmentPage 的 render 方法。EnvironmentPage 的 render 方法将会渲染页面元素。EnvironmentPage 的实现如代码清单 4-16 所示。



代码清单4-16 EnvironmentPage的实现

```
private[ui] class EnvironmentPage(parent: EnvironmentTab) extends WebUIPage("") {
  private val listener = parent.listener

  private def removePass(kv: (String, String)): (String, String) = {
    if (kv._1.toLowerCase.contains("password") || kv._1.toLowerCase.contains("secret")) {
      (kv._1, "******")
    } else kv
  }

  def render(request: HttpServletRequest): Seq[Node] = {
   // 调用UIUtils的listingTable方法生成JVM运行时信息、Spark属性信息、系统属性信息、类路径信
      息的表格
    val runtimeInformationTable = UIUtils.listingTable(
      propertyHeader, jvmRow, listener.jvmInformation, fixedWidth = true)
    val sparkPropertiesTable = UIUtils.listingTable(
      propertyHeader, propertyRow, listener.sparkProperties.map(removePass),
          fixedWidth = true)
    val systemPropertiesTable = UIUtils.listingTable(
      propertyHeader, propertyRow, listener.systemProperties, fixedWidth = true)
    val classpathEntriesTable = UIUtils.listingTable(
      classPathHeaders, classPathRow, listener.classpathEntries, fixedWidth = true)
    val content =
      <span>
        <h4>Runtime Information</h4> {runtimeInformationTable}
```

仅供非商业用途或交流学习使用


```
        <h4>Spark Properties</h4> {sparkPropertiesTable}
        <h4>System Properties</h4> {systemPropertiesTable}
        <h4>Classpath Entries</h4> {classpathEntriesTable}
      </span>
    // 调用UIUtils的headerSparkPage方法封装好css、js、header及页面布局等
    UIUtils.headerSparkPage("Environment", content, parent)
  }
    // 定义JVM运行时信息、Spark属性信息、系统属性信息的表格头部propertyHeader和类路径信息的表格头部
    // classPathHeaders
    private def propertyHeader = Seq("Name", "Value")
    private def classPathHeaders = Seq("Resource", "Source")
    // 定义JVM运行时信息的表格中每行数据的生成方法jvmRow
    private def jvmRow(kv: (String, String)) = <tr><td>{kv._1}</td><td>{kv._2}</
        td></tr>
    private def propertyRow(kv: (String, String)) = <tr><td>{kv._1}</
        td><td>{kv._2}</td></tr>
    private def classPathRow(data: (String, String)) = <tr><td>{data._1}</td><td>
        {data._2}</td></tr>
  }
```

根据代码清单 4-16，EnvironmentPage 的 render 方法利用从父节点 EnvironmentTab 中得到的 EnvironmentListener 中的统计监控数据生成 JVM 运行时、Spark 属性、系统属性及类路径等状态的摘要信息。以 JVM 运行时为例，页面渲染的步骤如下。

1）定义 JVM 运行时信息、Spark 属性信息、系统属性信息的表格头部 propertyHeader 和类路径信息的表格头部 classPathHeaders。

2）定义 JVM 运行时信息的表格中每行数据的生成方法 jvmRow。

3）调用 UIUtils 的 listingTable 方法生成 JVM 运行时信息、Spark 属性信息、系统属性信息、类路径信息的表格。

4）调用 UIUtils 的 headerSparkPage 方法封装好 CSS、JS、header 及页面布局等。

UIUtils 工具类的实现细节留给感兴趣的读者自行查阅，本文不再赘述。

4.4　创建心跳接收器

根据前文所述，生产环境的 Executor 运行于不同节点上。在 local 模式下 Driver 与 Executor 属于同一个进程，所以 Driver 与 Executor 可以直接使用本地调用交互，当 Executor 运行出现问题时，Driver 可以很方便地知道，例如，通过捕获异常。但在生产环境下，Driver 与 Executor 很可能不在同一个进程内，它们也许运行在不同的机器上，甚至在不同的机房里，因此 Driver 对 Executor 失去了掌控。为了能够掌控 Executor，在 Driver 中创建了这个心跳接收器。为了弄清楚心跳接收器，我们需要从心跳接收器的创建开始。

SparkContext 中创建 HeartbeatReceiver 的代码如下。

```
  _heartbeatReceiver = env.rpcEnv.setupEndpoint(
HeartbeatReceiver.ENDPOINT_NAME, new HeartbeatReceiver(this))
```

仅供非商业用途或交流学习使用

上面的代码中使用了 SparkEnv 的子组件 NettyRpcEnv 的 setupEndpoint 方法，此方法的作用是向 RpcEnv 的 Dispatcher 注册 HeartbeatReceiver，并返回 HeartbeatReceiver 的 NettyRpcEndpointRef 引用。有关 RpcEnv 及 NettyRpcEndpointRef 的内容将在 5.3 节详细介绍，而 HeartbeatReceiver 将在第 9 章详细介绍。

4.5 创建和启动调度系统

TaskScheduler 是 SparkContext 的重要组成部分，负责请求集群管理器给应用程序分配并运行 Executor（一级调度）和给任务分配 Executor 并运行任务（二级调度）。TaskScheduler 也可以看做任务调度的客户端。DAGScheduler 主要用于在任务正式交给 TaskSchedulerImpl 提交之前做一些准备工作，包括创建 Job、将 DAG 中的 RDD 划分到不同的 Stage、提交 Stage 等。代码清单 4-17 的代码用于在 SparkContext 中创建 TaskScheduler 和 DAGScheduler。

代码清单4-17 创建TaskScheduler和DAGScheduler

```
// Create and start the scheduler
val (sched, ts) = SparkContext.createTaskScheduler(this, master, deployMode)
_schedulerBackend = sched
_taskScheduler = ts
_dagScheduler = new DAGScheduler(this)
_heartbeatReceiver.ask[Boolean](TaskSchedulerIsSet)
```



根据代码清单 4-17，SparkContext 的 createTaskScheduler 方法用于创建任务调度器，此方法将返回 SchedulerBackend 和 TaskScheduler 的对偶，并由 _schedulerBackend 持有 SchedulerBackend 的引用，由 _taskScheduler 持有 TaskScheduler 的引用。代码清单 4-17 的最后还向 HeartbeatReceiver 发送 TaskSchedulerIsSet 消息，表示 SparkContext 的 _taskScheduler 属性已经持有了 TaskScheduler 的引用，HeartbeatReceiver 接收到 TaskSchedulerIsSet 消息后，将获取 SparkContext 的 _taskScheduler 属性设置到自身的 scheduler 属性中（将在第 9 章中详细介绍）。

createTaskScheduler 的实现如代码清单 4-18 所示。

代码清单4-18 创建任务调度器

```
private def createTaskScheduler(
    sc: SparkContext,
    master: String,
    deployMode: String): (SchedulerBackend, TaskScheduler) = {
  import SparkMasterRegex._

  // When running locally, don't try to re-execute tasks on failure.
  val MAX_LOCAL_TASK_FAILURES = 1

  master match {
```

仅供非商业用途或交流学习使用


```
case "local" =>
  val scheduler = new TaskSchedulerImpl(sc, MAX_LOCAL_TASK_FAILURES, isLocal = true)
  val backend = new LocalSchedulerBackend(sc.getConf, scheduler, 1)
  scheduler.initialize(backend)
  (backend, scheduler)

case LOCAL_N_REGEX(threads) =>
  def localCpuCount: Int = Runtime.getRuntime.availableProcessors()
  // local[*] estimates the number of cores on the machine; local[N] uses
      exactly N threads.
  val threadCount = if (threads == "*") localCpuCount else threads.toInt
  if (threadCount <= 0) {
    throw new SparkException(s"Asked to run locally with $threadCount threads")
  }
  val scheduler = new TaskSchedulerImpl(sc, MAX_LOCAL_TASK_FAILURES, isLocal = true)
  val backend = new LocalSchedulerBackend(sc.getConf, scheduler, threadCount)
  scheduler.initialize(backend)
  (backend, scheduler)

case LOCAL_N_FAILURES_REGEX(threads, maxFailures) =>
  def localCpuCount: Int = Runtime.getRuntime.availableProcessors()
  // local[*, M] means the number of cores on the computer with M failures
  // local[N, M] means exactly N threads with M failures
  val threadCount = if (threads == "*") localCpuCount else threads.toInt
  val scheduler = new TaskSchedulerImpl(sc, maxFailures.toInt, isLocal = true)
  val backend = new LocalSchedulerBackend(sc.getConf, scheduler, threadCount)
  scheduler.initialize(backend)
  (backend, scheduler)

case SPARK_REGEX(sparkUrl) =>
  val scheduler = new TaskSchedulerImpl(sc)
  val masterUrls = sparkUrl.split(",").map("spark://" + _)
  val backend = new StandaloneSchedulerBackend(scheduler, sc, masterUrls)
  scheduler.initialize(backend)
  (backend, scheduler)

case LOCAL_CLUSTER_REGEX(numSlaves, coresPerSlave, memoryPerSlave) =>
  // Check to make sure memory requested <= memoryPerSlave. Otherwise Spark
      will just hang.
  val memoryPerSlaveInt = memoryPerSlave.toInt
  if (sc.executorMemory > memoryPerSlaveInt) {
    throw new SparkException(
      "Asked to launch cluster with %d MB RAM / worker but requested %d MB/
          worker".format(
        memoryPerSlaveInt, sc.executorMemory))
  }

  val scheduler = new TaskSchedulerImpl(sc)
  val localCluster = new LocalSparkCluster(
    numSlaves.toInt, coresPerSlave.toInt, memoryPerSlaveInt, sc.conf)
  val masterUrls = localCluster.start()
  val backend = new StandaloneSchedulerBackend(scheduler, sc, masterUrls)
  scheduler.initialize(backend)
  backend.shutdownCallback = (backend: StandaloneSchedulerBackend) => {
    localCluster.stop()
  }
```



仅供非商业用途或交流学习使用


```
        (backend, scheduler)

      case masterUrl =>
        val cm = getClusterManager(masterUrl) match {
          case Some(clusterMgr) => clusterMgr
          case None => throw new SparkException("Could not parse Master URL: '" +
              master + "'")
        }
        try {
          val scheduler = cm.createTaskScheduler(sc, masterUrl)
          val backend = cm.createSchedulerBackend(sc, masterUrl, scheduler)
          cm.initialize(scheduler, backend)
          (backend, scheduler)
        } catch {
          case se: SparkException => throw se
          case NonFatal(e) =>
            throw new SparkException("External scheduler cannot be instantiated", e)
        }
    }
  }
```

根据代码清单 4-18，我们看到针对不同的部署模式（即 deployMode），创建任务调度器的方式也各有不同。将这一大段代码放置此处显得非常拖沓，但由于本书的内容将多次涉及此段代码，所以还是决定在此处展示。有关任务调度的详细内容将放在第 7 章详细介绍。

SparkContext 创建完 TaskScheduler 后，将启动 TaskScheduler，如代码清单 4-19 所示。

代码清单4-19 启动调度系统

```
_taskScheduler.start()
```

TaskScheduler 在启动的时候还会启动 DAGScheduler，这些内容也将在第 7 章详细介绍。

4.6 初始化块管理器 BlockManager

BlockManager 是 SparkEnv 中的组件之一，从命名来看应该是 Block 的管理器，实际上囊括了存储体系的所有组件和功能，是存储体系中最重要的组件。SparkContext 初始化的过程中会对 BlockManager 进行初始化，代码如下。

```
_env.blockManager.initialize(_applicationId)
```

_applicationId 是应用程序向 Master 注册时，Master 为其创建的应用标识。BlockManager 的初始化将在 6.7 节详细介绍。

4.7 启动度量系统

3.4 节曾介绍过度量系统中的 Source 及 Sink，但是并未将整个度量系统贯穿起来。Metrics-

仅供非商业用途或交流学习使用

System 对 Source 和 Sink 进行封装，将 Source 的数据输出到不同的 Sink。MetricsSystem 是 SparkEnv 内部的组件之一，是整个 Spark 应用程序的度量系统。启动度量系统的代码如下。

```
_env.metricsSystem.start()
_env.metricsSystem.getServletHandlers.foreach(handler => ui.foreach(_.
    attachHandler(handler)))
```

上述代码还调用 WebUI 的 attachHandler 方法（见代码清单 4-4）将度量系统的 ServletContextHandler 添加到了 Spark UI 中。有关启动度量系统和 MetricsSystem 的 getServletHandlers 方法将在 5.8 节详细介绍。

4.8　创建事件日志监听器

EventLoggingListener 是将事件持久化到存储的监听器，是 SparkContext 中的可选组件。当 spark.eventLog.enabled 属性为 true 时启用。这里来看看 SparkContext 是如何创建 EventLoggingListener 的。

```
_eventLogger =
if (isEventLogEnabled) {
  val logger =
    new EventLoggingListener(_applicationId, _applicationAttemptId, _
      eventLogDir.get,
      _conf, _hadoopConfiguration)
  logger.start()
  listenerBus.addListener(logger)
  Some(logger)
} else {
  None
}
```

EventLoggingListener 也将参与到对事件总线中事件的监听中，并把感兴趣的事件记录到日志。

以 EventLoggingListener 感兴趣的 SparkListenerBlockManagerRemoved 事件为例，EventLoggingListener 重写的 onBlockManagerRemoved 方法将对 SparkListenerBlockManagerRemoved 事件进行处理，代码如下。

```
override def onBlockManagerRemoved(event: SparkListenerBlockManagerRemoved): Unit
    = {
  logEvent(event, flushLogger = true)
}
```

EventLoggingListener 最为核心的方法是 logEvent，其实现如下。

```
private def logEvent(event: SparkListenerEvent, flushLogger: Boolean = false) {
  val eventJson = JsonProtocol.sparkEventToJson(event)
  writer.foreach(_.println(compact(render(eventJson))))
  if (flushLogger) {
```

仅供非商业用途或交流学习使用


```
      writer.foreach(_.flush())
      hadoopDataStream.foreach(_.hflush())
    }
    if (testing) {
      loggedEvents += eventJson
    }
  }
```

logEvent 用于将事件转换为 Json 字符串后写入日志文件。

4.9 创建和启动 ExecutorAllocationManager

ExecutorAllocationManager 的作用已在 4.1 节有过介绍，更为准确地说，ExecutorAllocationManager 是基于工作负载动态分配和删除 Executor 的代理。简单来讲，ExecutorAllocationManager 与集群管理器之间的关系可以用图 4-3 来表示。

ExecutorAllocationManager 内部会定时根据工作负载计算所需的 Executor 数量，如果对 Executor 需求数量大于之前向集群管理器申请的 Executor 数量，那么向集群管理器申请添加 Executor；如果对 Executor 需求数量小于之前向集群管理器申请的 Executor 数量，那么向集群管理器申请取消部分 Executor。此外，ExecutorAllocationManager 内部还会定时向集群管理器申请移除（“杀死”）过期的 Executor。

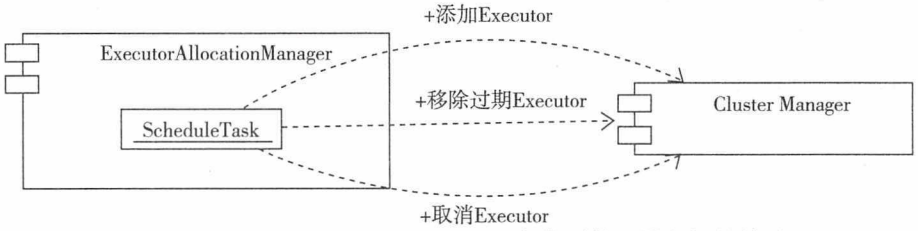


图 4-3 ExecutorAllocationManager 与集群管理器之间的关系

有了对 ExecutorAllocationManager 的了解，这里来看看 SparkContext 是如何创建和启动 ExecutorAllocationManager 的。创建 ExecutorAllocationManager 的代码如下。

```
val dynamicAllocationEnabled = Utils.isDynamicAllocationEnabled(_conf)
_executorAllocationManager =
  if (dynamicAllocationEnabled) {
    schedulerBackend match {
      case b: ExecutorAllocationClient =>
        Some(new ExecutorAllocationManager(
          schedulerBackend.asInstanceOf[ExecutorAllocationClient], listenerBus, _conf))
      case _ =>
        None
    }
  } else {
    None
  }
_executorAllocationManager.foreach(_.start())
```

仅供非商业用途或交流学习使用

根据上述代码，创建 ExecutorAllocationManager 的步骤如下。

1）调用 Utils 工具类的 isDynamicAllocationEnabled 方法判断是否需要启用 ExecutorAllocationManager。根据附录 A 中对 isDynamicAllocationEnabled 方法的介绍，我们知道在配置 spark.dynamicAllocation.enabled 属性为 true 的前提下，在非 Local 模式下或者当 spark.dynamicAllocation.testing 属性为 true 时启用 ExecutorAllocationManager。

2）在 SchedulerBackend 的实现类同时实现了特质 ExecutorAllocationClient 的情况下，才会创建 ExecutorAllocationManager。

3）调用 ExecutorAllocationManager 的 start 方法启动 ExecutorAllocationManager。

ExecutorAllocationManager 的 start 方法的实现如下。

```
def start(): Unit = {
  listenerBus.addListener(listener)
  val scheduleTask = new Runnable() {
    override def run(): Unit = {
      try {
        schedule()
      } catch {
        case ct: ControlThrowable =>
          throw ct
        case t: Throwable =>
          logWarning(s"Uncaught exception in thread ${Thread.currentThread().
              getName}", t)
      }
    }
  }
   executor.scheduleWithFixedDelay(scheduleTask, 0, intervalMillis, TimeUnit.
MILLISECONDS)
  client.requestTotalExecutors(numExecutorsTarget, localityAwareTasks, hostTo
      LocalTaskCount)
}
```

根据上述代码，ExecutorAllocationManager 的 start 方法的执行步骤如下。

1）向事件总线添加 ExecutorAllocationListener。

2）创建定时调度的任务 scheduleTask，此任务主要调用 schedule 方法。

3）将 scheduleTask 提交给 executor（executor 是只有一个线程的 ScheduledThreadPoolExecutor），以固定的间隔 intervalMillis（值为 100）进行调度。

4）调用 ExecutorAllocationClient 的 requestTotalExecutors 方法请求所有的 Executor。numExecutorsTarget 是动态分配 Executor 的总数，取 spark.dynamicAllocation.initialExecutors、spark.dynamicAllocation.minExecutors、spark.executor.instances 三个属性配置的最大值。localityAwareTasks 是由本地性偏好的 Task 数量。hostToLocalTaskCount 是 Host 与想要在此节点上运行的 Task 的数量之间的映射关系。



在 SchedulerBackend 的实现类中只有 CoarseGrainedSchedulerBackend 同时实现了特质 ExecutorAllocationClient，因此对 CoarseGrainedSchedulerBackend 实现的 request-

仅供非商业用途或交流学习使用

TotalExecutors 方法的具体内容将放在第 9 章详细介绍。

定时任务 scheduleTask 会按照固定的时间间隔调用 ExecutorAllocationManager 的 schedule 方法，以调整待执行 Executor 请求的数量和运行的 Executor 的数量。schedule 方法的实现如下。

```
private def schedule(): Unit = synchronized {
  val now = clock.getTimeMillis
  updateAndSyncNumExecutorsTarget(now)
  val executorIdsToBeRemoved = ArrayBuffer[String]()
  removeTimes.retain { case (executorId, expireTime) =>
    val expired = now >= expireTime
    if (expired) {
      initializing = false
      executorIdsToBeRemoved += executorId
    }
    !expired
  }
  if (executorIdsToBeRemoved.nonEmpty) {
    removeExecutors(executorIdsToBeRemoved)
  }
}
```

根据上述代码，schedule 方法的执行步骤如下。

1）调用 updateAndSyncNumExecutorsTarget 方法重新计算所需的 Executor 数量，并更新请求的 Executor 数量。

2）对过期的 Executor 进行删除。removeExecutors 方法将利用 ExecutorAllocationClient 的 killExecutors 方法通知集群管理器“杀死”Executor。killExecutors 方法需要 ExecutorAllocationClient 的实现类去实现。

updateAndSyncNumExecutorsTarget 方法的实现如下。

```
private def updateAndSyncNumExecutorsTarget(now: Long): Int = synchronized {
 val maxNeeded = maxNumExecutorsNeeded // 获得实际需要的Executor的最大数量maxNeeded

  if (initializing) { // ExecutorAllocationManager还在初始化，则返回0
    0
  } else if (maxNeeded < numExecutorsTarget) { // 减少需要的Executor的数量
    val oldNumExecutorsTarget = numExecutorsTarget
    numExecutorsTarget = math.max(maxNeeded, minNumExecutors)
    numExecutorsToAdd = 1

    if (numExecutorsTarget < oldNumExecutorsTarget) {
        client.requestTotalExecutors(numExecutorsTarget, localityAwareTasks,
hostToLocalTaskCount)
        logDebug(s"Lowering target number of executors to $numExecutorsTarget
(previously " +
          s"$oldNumExecutorsTarget) because not all requested executors are
actually needed")
    }
    numExecutorsTarget - oldNumExecutorsTarget
  } else if (addTime != NOT_SET && now >= addTime) { // 添加Executor
```

仅供非商业用途或交流学习使用


```
      val delta = addExecutors(maxNeeded)
      logDebug(s"Starting timer to add more executors (to " +
        s"expire in $sustainedSchedulerBacklogTimeoutS seconds)")
      addTime += sustainedSchedulerBacklogTimeoutS * 1000
      delta
    } else {
      0
    }
  }
```

根据上述代码，updateAndSyncNumExecutorsTarget 方法的执行步骤如下。

1）调用 maxNumExecutorsNeeded 方法获得实际需要的 Executor 的最大数量 maxNeeded。

2）如果 ExecutorAllocationManager 还在初始化，则返回 0。

3）如果 Executor 的目标数量（numExecutorsTarget）超过我们实际需要的数量（maxNeeded），那么首先将 numExecutorsTarget 设置为 maxNeeded 与最小 Executor 数量（minNumExecutors）之间的最大值，然后调用 ExecutorAllocationClient 的 requestTotalExecutors 方法重新请求 numExecutorsTarget 指定的目标 Executor 数量，以此停止添加新的执行程序，并通知集群管理器取消额外的待处理 Executor 的请求，最后返回减少的 Executor 数量。

4）如果 maxNeeded 大于等于 numExecutorsTarget，且当前时间大于上次添加 Executor 的时间，那么首先调用 addExecutors 方法[⊖]通知集群管理器添加额外的 Executor，然后更新添加 Executor 的时间，最后返回添加的 Executor 数量。

这里对 ExecutorAllocationManager 的主要工作原理进行了分析，maxNumExecutorsNeeded 方法、addExecutors 方法及 removeExecutors 方法的实现留给感兴趣的读者自行阅读。local-cluster 和 Standalone 模式下的集群管理器 Master 如何给应用程序分配或取消 Executor 的分析，将在 9.9.5 节详细介绍。

根据对 ExecutorAllocationManager 的分析，可以用图 4-4 来表示 Executor 的动态分配过程。

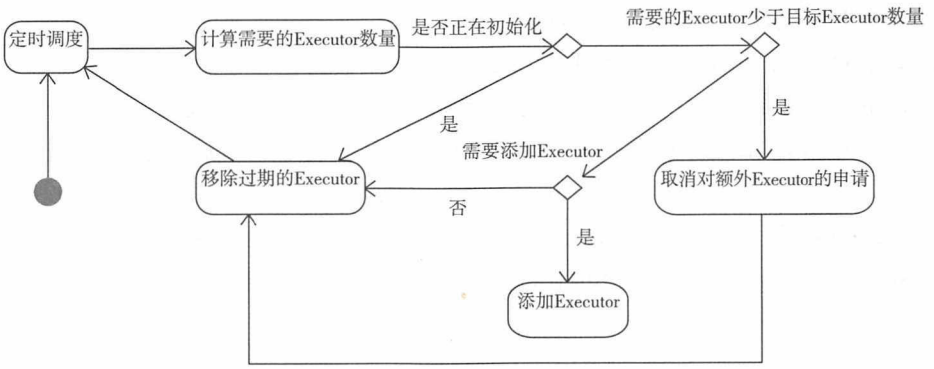


图 4-4　Executor 的动态分配过程

⊖ 此方法也是利用 ExecutorAllocationClient 的 requestTotalExecutors 方法。

仅供非商业用途或交流学习使用

4.10 ContextCleaner 的创建与启动

ContextCleaner 用于清理那些超出应用范围的 RDD、Shuffle 对应的 map 任务状态、Shuffle 元数据、Broadcast 对象及 RDD 的 Checkpoint 数据。

4.10.1 创建 ContextCleaner

创建 ContextCleaner 的代码如下。

```
_cleaner =
  if (_conf.getBoolean("spark.cleaner.referenceTracking", true)) {
    Some(new ContextCleaner(this))
  } else {
    None
  }
_cleaner.foreach(_.start())
```

根据上述代码，我们知道可以通过配置属性 spark.cleaner.referenceTracking（默认是 true）来决定是否启用 ContextCleaner。

ContextCleaner 的组成如下。

- referenceQueue：缓存顶级的 AnyRef 引用。
- referenceBuffer：缓存 AnyRef 的虚引用。
- listeners：缓存清理工作的监听器数组。
- cleaningThread：用于具体清理工作的线程。此线程为守护线程，名称为 Spark Context Cleaner。
- periodicGCService：类型为 ScheduledExecutorService，用于执行 GC（Garbage Collection，垃圾收集）的调度线程池，此线程池只包含一个线程，启动的线程名称以 context-cleaner-periodic-gc 开头。
- periodicGCInterval：执行 GC 的时间间隔。可通过 spark.cleaner.periodicGC.interval 属性进行配置，默认是 30 分钟。
- blockOnCleanupTasks：清理非 Shuffle 的其他数据是否是阻塞式的。可通过 spark.cleaner.referenceTracking.blocking 属性进行配置，默认是 true。
- blockOnShuffleCleanupTasks：清理 Shuffle 数据是否是阻塞式的。可通过 spark.cleaner.referenceTracking.blocking.shuffle 属性进行配置，默认是 false。清理 Shuffle 数据包括清理 MapOutputTracker 中指定 ShuffleId 对应的 map 任务状态和 ShuffleManager 中注册的 ShuffleId 对应的 Shuffle 元数据。
- stopped：ContextCleaner 是否停止的状态标记。



4.10.2 启动 ContextCleaner

SparkContext 在初始化的过程中会启动 ContextCleaner，只有这样 ContextCleaner

仅供非商业用途或交流学习使用

才能够清理那些超出应用范围的 RDD、Shuffle 对应的 map 任务状态、Shuffle 元数据、Broadcast 对象及 RDD 的 Checkpoint 数据。启动 ContextCleaner 的代码如下。

```
def start(): Unit = {
  cleaningThread.setDaemon(true)
  cleaningThread.setName("Spark Context Cleaner")
  cleaningThread.start()
  periodicGCService.scheduleAtFixedRate(new Runnable {
    override def run(): Unit = System.gc()
  }, periodicGCInterval, periodicGCInterval, TimeUnit.SECONDS)
}
```

根据上述代码，启动 ContextCleaner 的步骤如下。

1）将 cleaningThread 设置为守护线程，并指定名称为 Spark Context Cleaner。

2）启动 cleaningThread。

3）给 periodicGCService 设置以 periodicGCInterval 作为时间间隔定时进行 GC 操作的任务。

除了 GC 的定时器，ContextCleaner 其余部分的工作原理和 listenerBus 一样（也采用监听器模式，由异步线程来处理），因此不再赘述。异步线程实际只是调用 keepCleaning 方法，其实现如代码清单 4-20 所示。

代码清单4-20　keepCleaning的实现



```
private def keepCleaning(): Unit = Utils.tryOrStopSparkContext(sc) {
  while (!stopped) {
    try {
      val reference = Option(referenceQueue.remove(ContextCleaner.REF_QUEUE_POLL_
          TIMEOUT))
        .map(_.asInstanceOf[CleanupTaskWeakReference])
      synchronized {
        reference.foreach { ref =>
          logDebug("Got cleaning task " + ref.task)
          referenceBuffer.remove(ref)
          ref.task match {
            case CleanRDD(rddId) =>
              doCleanupRDD(rddId, blocking = blockOnCleanupTasks)
            case CleanShuffle(shuffleId) =>
              doCleanupShuffle(shuffleId, blocking = blockOnShuffleCleanupTasks)
            case CleanBroadcast(broadcastId) =>
              doCleanupBroadcast(broadcastId, blocking = blockOnCleanupTasks)
            case CleanAccum(accId) =>
              doCleanupAccum(accId, blocking = blockOnCleanupTasks)
            case CleanCheckpoint(rddId) =>
              doCleanCheckpoint(rddId)
          }
        }
      }
    } catch {
      case ie: InterruptedException if stopped => // ignore
      case e: Exception => logError("Error in cleaning thread", e)
    }
```

仅供非商业用途或交流学习使用


```
  }
}
```

从代码清单 4-20 可以看出，异步线程将匹配各种引用，并执行相应的方法进行清理。以 doCleanupRDD 为例，其实现如代码清单 4-21 所示。

代码清单4-21 清理RDD数据

```
def doCleanupRDD(rddId: Int, blocking: Boolean): Unit = {
  try {
    logDebug("Cleaning RDD " + rddId)
    sc.unpersistRDD(rddId, blocking)
    listeners.asScala.foreach(_.rddCleaned(rddId))
    logInfo("Cleaned RDD " + rddId)
  } catch {
    case e: Exception => logError("Error cleaning RDD " + rddId, e)
  }
}
```

根据代码清单 4-21，doCleanupRDD 的执行步骤如下。

1）调用 SparkContext 的 unpersistRDD 方法从内存或磁盘中移除 RDD。

2）从 persistentRdds 中移除对 RDD 的跟踪。

3）调用所有监听器的 rddCleaned 方法。



4.11 额外的 SparkListener 与启动事件总线

SparkContext 中提供了添加用于自定义 SparkListener 的地方，代码如下。

```
setupAndStartListenerBus()
```

setupAndStartListenerBus 方法的实现如下。

```
private def setupAndStartListenerBus(): Unit = {
  try {
    // 获取用户自定义的SparkListener的类名
    val listenerClassNames: Seq[String] =
      conf.get("spark.extraListeners", "").split(',').map(_.trim).filter(_ != "")
    // 通过反射生成每一个自定义SparkListener的实例，并添加到事件总线的监听器列表中
    for (className <- listenerClassNames) {
      // Use reflection to find the right constructor
      val constructors = {
        val listenerClass = Utils.classForName(className)
        listenerClass
            .getConstructors
            .asInstanceOf[Array[Constructor[_ <: SparkListenerInterface]]]
      }
      val constructorTakingSparkConf = constructors.find { c =>
        c.getParameterTypes.sameElements(Array(classOf[SparkConf]))
      }
      lazy val zeroArgumentConstructor = constructors.find { c =>
```

仅供非商业用途或交流学习使用


```
            c.getParameterTypes.isEmpty
          }
          val listener: SparkListenerInterface = {
            if (constructorTakingSparkConf.isDefined) {
              constructorTakingSparkConf.get.newInstance(conf)
            } else if (zeroArgumentConstructor.isDefined) {
              zeroArgumentConstructor.get.newInstance()
            } else {
              throw new SparkException(
                s"$className did not have a zero-argument constructor or a" +
                  " single-argument constructor that accepts SparkConf. Note: if the
                      class is" +
                  " defined inside of another Scala class, then its constructors may
                      accept an" +
                  " implicit parameter that references the enclosing class; in this
                      case, you must" +
                  " define the listener as a top-level class in order to prevent this
                      extra" +
                  " parameter from breaking Spark's ability to find a valid const-
                      ructor.")
            }
          }
          listenerBus.addListener(listener)
          logInfo(s"Registered listener $className")
        }
      } catch {
        case e: Exception =>
          try {
            stop()
          } finally {
            throw new SparkException(s"Exception when registering SparkListener", e)
          }
      }
      // 启动事件总线，并将_listenerBusStarted置为true
      listenerBus.start()
      _listenerBusStarted = true
    }
```

根据上述代码，setupAndStartListenerBus 的执行步骤如下。

1）从 spark.extraListeners 属性中获取用户自定义的 SparkListener 的类名。用户可以通过逗号分隔多个自定义 SparkListener。

2）通过反射生成每一个自定义 SparkListener 的实例，并添加到事件总线的监听器列表中。

3）启动事件总线，并将 _listenerBusStarted 置为 true。

4.12　Spark 环境更新

用户提交任务时往往需要添加额外的 Jar 包或其他文件，用户任务的执行将依赖这些文件。这些文件该如何指定？任务在各个节点上运行时又是如何获取到这些文件的呢？我们

仅供非商业用途或交流学习使用

首先回答第一个问题。

在 SparkContext 的初始化过程中会读取用户指定的 Jar 文件或其他文件，代码如下。

```
_jars = Utils.getUserJars(_conf)
_files = _conf.getOption("spark.files").map(_.split(",")).map(_.filter(_.
    nonEmpty))
  .toSeq.flatten
```

上述代码首先读取用户设置的 Jar 文件，然后读取用户设置的其他文件。当用户选择的部署模式是 YARN 时，_jars 是由 spark.jars 属性指定的 Jar 文件和 spark.yarn.dist.jars 属性指定的 Jar 文件的并集。其他模式下只采用由 spark.jars 属性指定的 Jar 文件。这里使用了 Utils 的 getUserJars 方法，其具体介绍请阅读附录 A。通过 spark.files 属性可以指定其他文件。

回答了第一个问题，第二个问题该如何解决？

在 SparkContext 的初始化过程中有以下代码。

```
def jars: Seq[String] = _jars
def files: Seq[String] = _files
  // Add each JAR given through the constructor
  if (jars != null) {
    jars.foreach(addJar)
  }
  if (files != null) {
    files.foreach(addFile)
  }
```



上述代码中，jars 和 files 是两个简单的方法，分别用来获取 Jar 包的序列集合和其他文件的序列集合。上述代码还遍历每一个 Jar 文件并调用 addJar 方法，遍历每一个其他文件并调用 addFile 方法。

addJar 方法是做什么的呢？它用于将 Jar 文件添加到 Driver 的 RPC 环境中。addJar 的实现如代码清单 4-22 所示。

代码清单4-22　addJar的实现

```
def addJar(path: String) {
  if (path == null) {
    logWarning("null specified as parameter to addJar")
  } else {
    var key = ""
    if (path.contains("\\")) {
      key = env.rpcEnv.fileServer.addJar(new File(path))
    } else {
      val uri = new URI(path)
      Utils.validateURL(uri)
      key = uri.getScheme match {
        case null | "file" =>
          try {
            env.rpcEnv.fileServer.addJar(new File(uri.getPath))
          } catch {
```

仅供非商业用途或交流学习使用


```
            case exc: FileNotFoundException =>
              logError(s"Jar not found at $path")
              null
          }
        case "local" =>
          "file:" + uri.getPath
        case _ =>
          path
      }
    }
    if (key != null) {
      val timestamp = System.currentTimeMillis
      if (addedJars.putIfAbsent(key, timestamp).isEmpty) {
        logInfo(s"Added JAR $path at $key with timestamp $timestamp")
        postEnvironmentUpdate()
      }
    }
  }
}
```

根据代码清单 4-22，将调用 SparkEnv 的 RpcEnv 的 fileServer（fileServer 实际是 5.3.5 节要介绍的 NettyStreamManager）的 addJar 方法把 Jar 文件添加到 Driver 本地 RpcEnv 的 NettyStreamManager 中，并将 Jar 文件添加的时间戳信息缓存到 addedJars 中。SparkEnv 及 fileServer 的内容将在第 5 章详细介绍。

addFile 与 addJar 类似，其实现如代码清单 4-23 所示。



代码清单4-23　addFile的实现

```
def addFile(path: String): Unit = {
  addFile(path, false)
}
def addFile(path: String, recursive: Boolean): Unit = {
  val uri = new Path(path).toUri
  val schemeCorrectedPath = uri.getScheme match {
    case null | "local" => new File(path).getCanonicalFile.toURI.toString
    case _ => path
  }
  val hadoopPath = new Path(schemeCorrectedPath)
  val scheme = new URI(schemeCorrectedPath).getScheme
  if (!Array("http", "https", "ftp").contains(scheme)) {
    val fs = hadoopPath.getFileSystem(hadoopConfiguration)
    val isDir = fs.getFileStatus(hadoopPath).isDirectory
    if (!isLocal && scheme == "file" && isDir) {
      throw new SparkException(s"addFile does not support local directories when
          not running " +
        "local mode.")
    }
    if (!recursive && isDir) {
        throw new SparkException(s"Added file $hadoopPath is a directory and
recursive is not " +
        "turned on.")
    }
  } else {
```

仅供非商业用途或交流学习使用


```
      Utils.validateURL(uri)
    }

    val key = if (!isLocal && scheme == "file") {
      env.rpcEnv.fileServer.addFile(new File(uri.getPath))
    } else {
      schemeCorrectedPath
    }
    val timestamp = System.currentTimeMillis
    if (addedFiles.putIfAbsent(key, timestamp).isEmpty) {
      logInfo(s"Added file $path at $key with timestamp $timestamp")
      Utils.fetchFile(uri.toString, new File(SparkFiles.getRootDirectory()), conf,
        env.securityManager, hadoopConfiguration, timestamp, useCache = false)
      postEnvironmentUpdate()
    }
  }
```

根据代码清单 4-23，将调用 SparkEnv 的 RpcEnv 的 fileServer 的 addFile 方法将文件添加到 Driver 本地 RpcEnv 的 NettyStreamManager 中，并将文件添加的时间戳信息缓存到 addedFiles 中。

通过 addJar 和 addFile 可以将各种任务执行所依赖的文件添加到 Driver 的 RPC 环境中，这样各个 Executor 节点就可以使用 RPC 从 Driver 将文件下载到本地，以供任务执行。

在 addJar 和 addFile 方法的最后都调用了 postEnvironmentUpdate 方法，而且在 SparkContext 初始化过程的最后也会调用 postEnvironmentUpdate，代码如下。

```
postEnvironmentUpdate ()
```

由于 addJar 和 addFile 可能会对应用的环境产生影响，所以在 SparkContext 初始化过程的最后需要调用 postEnvironmentUpdate 方法更新环境。postEnvironmentUpdate 的实现如代码清单 4-24 所示。

代码清单4-24　postEnvironmentUpdate的实现

```
private def postEnvironmentUpdate() {
  if (taskScheduler != null) {
    val schedulingMode = getSchedulingMode.toString
    val addedJarPaths = addedJars.keys.toSeq
    val addedFilePaths = addedFiles.keys.toSeq
    // 将JVM参数、Spark 属性、系统属性、classPath等信息设置为环境明细信息
    val environmentDetails = SparkEnv.environmentDetails(conf, schedulingMode,
       addedJarPaths,addedFilePaths)
    // 生成SparkListenerEnvironmentUpdate事件，并投递到事件总线
    val environmentUpdate = SparkListenerEnvironmentUpdate(environmentDetails)
    listenerBus.post(environmentUpdate)
  }
}
```

根据代码清单 4-24，postEnvironmentUpdate 的处理步骤如下。

1）通过调用 SparkEnv 的方法 environmentDetails（见代码清单 4-25），将环境的 JVM

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

参数、Spark 属性、系统属性、classPath 等信息设置为环境明细信息。

2）生成事件 SparkListenerEnvironmentUpdate（此事件携带环境明细信息），并投递到事件总线 listenerBus，此事件最终将被 EnvironmentListener 监听，并影响 EnvironmentPage 页面中的输出内容。

代码清单4-25　投递环境更新事件

```
private[spark]
def environmentDetails(
    conf: SparkConf,
    schedulingMode: String,
    addedJars: Seq[String],
    addedFiles: Seq[String]): Map[String, Seq[(String, String)]] = {
  import Properties._
  val jvmInformation = Seq(
    ("Java Version", s"$javaVersion ($javaVendor)"),
    ("Java Home", javaHome),
    ("Scala Version", versionString)
  ).sorted
  val schedulerMode =
    if (!conf.contains("spark.scheduler.mode")) {
      Seq(("spark.scheduler.mode", schedulingMode))
    } else {
      Seq[(String, String)]()
    }
  val sparkProperties = (conf.getAll ++ schedulerMode).sorted

  val systemProperties = Utils.getSystemProperties.toSeq
  val otherProperties = systemProperties.filter { case (k, _) =>
    k != "java.class.path" && !k.startsWith("spark.")
  }.sorted

  val classPathEntries = javaClassPath
    .split(File.pathSeparator)
    .filterNot(_.isEmpty)
    .map((_, "System Classpath"))
  val addedJarsAndFiles = (addedJars ++ addedFiles).map((_, "Added By User"))
  val classPaths = (addedJarsAndFiles ++ classPathEntries).sorted

  Map[String, Seq[(String, String)]](
    "JVM Information" -> jvmInformation,
    "Spark Properties" -> sparkProperties,
    "System Properties" -> otherProperties,
    "Classpath Entries" -> classPaths)
}
```

4.13　SparkContext 初始化的收尾

在 SparkContext 初始化过程的最后，有一些收尾工作要做，代码如下。

```
postApplicationStart() // 向事件总线投递SparkListenerApplicationStart事件
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
    _taskScheduler.postStartHook() // 等待SchedulerBackend准备完成
    // 向度量系统注册Source
    _env.metricsSystem.registerSource(_dagScheduler.metricsSource)
    _env.metricsSystem.registerSource(new BlockManagerSource(_env.blockManager))
    _executorAllocationManager.foreach { e =>
      _env.metricsSystem.registerSource(e.executorAllocationManagerSource)
    }
    // 添加SparkContext的关闭钩子
    logDebug("Adding shutdown hook") // force eager creation of logger
    _shutdownHookRef = ShutdownHookManager.addShutdownHook(
      ShutdownHookManager.SPARK_CONTEXT_SHUTDOWN_PRIORITY) { () =>
      logInfo("Invoking stop() from shutdown hook")
      stop()
  }
  // 将SparkContext标记为激活
  SparkContext.setActiveContext(this, allowMultipleContexts)
```

根据上述代码，收尾的工作包括以下几项。

1）调用 postApplicationStart 方法（见代码清单 4-26），向事件总线投递 SparkListener-ApplicationStart（即应用启动事件）。

代码清单4-26　postApplicationStart的实现

```
private def postApplicationStart() {
  listenerBus.post(SparkListenerApplicationStart(appName, Some(applicationId),
      startTime, sparkUser, applicationAttemptId, schedulerBackend.
getDriverLogUrls))
}
```



2）调用 TaskScheduler 的 postStartHook 方法（此方法的实现见代码清单 7-108），等待 SchedulerBackend 准备完成。

3）向度量系统注册 DAGSchedulerSource、BlockManagerSource 及 ExecutorAllocation ManagerSource。

4）添加 SparkContext 的关闭钩子，使得 JVM 退出之前调用 SparkContext 的 stop 方法进行一些关闭工作。

5）将 SparkContext 标记为激活。setActiveContext 方法的实现如代码清单 4-31 所示。

4.14　SparkContext 提供的常用方法

1. broadcast

broadcast 方法用于广播给定的对象，其实现如代码清单 4-27 所示。

代码清单4-27　广播给定对象

```
def broadcast[T: ClassTag](value: T): Broadcast[T] = {
  assertNotStopped()
  require(!classOf[RDD[_]].isAssignableFrom(classTag[T].runtimeClass),
    "Can not directly broadcast RDDs; instead, call collect() and broadcast the
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
        result.")
    val bc = env.broadcastManager.newBroadcast[T](value, isLocal)
    val callSite = getCallSite
    logInfo("Created broadcast " + bc.id + " from " + callSite.shortForm)
    cleaner.foreach(_.registerBroadcastForCleanup(bc))
    bc
  }
```

根据代码清单 4-27，broadcast 的实质是调用了 SparkEnv 的子组件 BroadcastManager 的 newBroadcast 方法生成广播对象。BroadcastManager 及其 newBroadcast 方法将在第 5 章详细介绍。

2. addSparkListener

addSparkListener 方法（见代码清单 4-28）用于向 LiveListenerBus 中添加实现了特质 SparkListenerInterface 的监听器。

代码清单4-28　添加SparkListener

```
@DeveloperApi
def addSparkListener(listener: SparkListenerInterface) {
  listenerBus.addListener(listener)
}
```

3. 多种多样的 runJob

SparkContext 提供了多个重载的 runJob 方法，但是这些 runJob 方法最终都将调用代码清单 4-29 所示的 runJob 方法。

代码清单4-29　runJob方法

```
def runJob[T, U: ClassTag](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    resultHandler: (Int, U) => Unit): Unit = {
  if (stopped.get()) {
    throw new IllegalStateException("SparkContext has been shutdown")
  }
  val callSite = getCallSite
  val cleanedFunc = clean(func)
  logInfo("Starting job: " + callSite.shortForm)
  if (conf.getBoolean("spark.logLineage", false)) {
    logInfo("RDD's recursive dependencies:\n" + rdd.toDebugString)
  }
  // 将DAG及RDD提交给DAGScheduler进行调度
  dagScheduler.runJob(rdd, cleanedFunc, partitions, callSite, resultHandler,
      localProperties.get)
  progressBar.foreach(_.finishAll())
  rdd.doCheckpoint() // 保存检查点
}
```

根据代码清单 4-29，runJob 方法的执行步骤如下。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

1）调用 DAGScheduler 的 runJob 方法将已经构建好 DAG 的 RDD 提交给 DAGScheduler 进行调度。DAGScheduler 将在 7.4 节进行详细介绍。

2）调用 RDD 的 doCheckpoint 方法（见代码清单 10-25）保存检查点。

4. setCheckpointDir

SparkContext 的 setCheckpointDir 方法（见代码清单 4-30）用于给作业中的 RDD 指定检查点保存的目录。指定检查点目录是启用检查点机制的前提。本书将在 10.3.3 节详细介绍检查点的实现。

代码清单4-30 指定检查点保存的目录

```
def setCheckpointDir(directory: String) {
  if (!isLocal && Utils.nonLocalPaths(directory).isEmpty) {
      logWarning("Spark is not running in local mode, therefore the checkpoint
directory " +
       s"must not be on the local filesystem. Directory '$directory' " +
       "appears to be on the local filesystem.")
  }
  checkpointDir = Option(directory).map { dir =>
    val path = new Path(dir, UUID.randomUUID().toString)
    val fs = path.getFileSystem(hadoopConfiguration)
    fs.mkdirs(path)
    fs.getFileStatus(path).getPath.toString
  }
}
```



4.15 SparkContext 的伴生对象

SparkContext 的伴生对象中提供了很多常用的功能，所以有必要对它们进行分析。SparkContext 的伴生对象中包含以下属性。

- VALID_LOG_LEVELS：有效的日志级别。VALID_LOG_LEVELS 包括 ALL、DEBUG、ERROR、FATAL、INFO、OFF、TRACE、WARN 等。
- SPARK_CONTEXT_CONSTRUCTOR_LOCK：对 SparkContext 进行构造时使用的锁，以此保证构造 SparkContext 的过程是线程安全的。
- activeContext：类型为 AtomicReference[SparkContext]，用于保存激活的 SparkContext。
- contextBeingConstructed：标记当前正在构造 SparkContext。

除了这些属性，SparkContext 的伴生对象还提供了一些常用的方法，下面一一进行介绍。

1. setActiveContext

setActiveContext 方法（见代码清单 4-31）用于将 SparkContext 作为激活的 SparkContext，保存到 activeContext 中。

書籍是人类进步的阶梯

仅供非商业用途或交流学习使用

代码清单4-31　设置激活的SparkContext

```
private[spark] def setActiveContext(
    sc: SparkContext,
    allowMultipleContexts: Boolean): Unit = {
  SPARK_CONTEXT_CONSTRUCTOR_LOCK.synchronized {
    assertNoOtherContextIsRunning(sc, allowMultipleContexts)
    contextBeingConstructed = None
    activeContext.set(sc)
  }
}
```

2. getOrCreate(config: SparkConf)

此方法（见代码清单 4-32）的功能为：如果没有激活的 SparkContext，则构造 SparkContext，并调用 setActiveContext 方法保存到 activeContext 中，最后返回激活的 SparkContext。

代码清单4-32　获取激活的SparkContext

```
def getOrCreate(config: SparkConf): SparkContext = {
  SPARK_CONTEXT_CONSTRUCTOR_LOCK.synchronized {
    if (activeContext.get() == null) {
      setActiveContext(new SparkContext(config), allowMultipleContexts = false)
    } else {
      if (config.getAll.nonEmpty) {
        logWarning("Using an existing SparkContext; some configuration may not
            take effect.")
      }
    }
    activeContext.get()
  }
}
```



SparkContext 伴生对象中还提供了无参的 getOrCreate 方法，其实现与 getOrCreate(config: SparkConf) 方法类似。

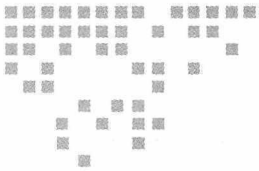
4.16　小结

本章首先介绍了 SparkContext 的作用及其组成，然后按照执行顺序介绍 SparkContext 的初始化过程。初始化过程中需要创建并启动很多组件，例如，SparkEnv、SparkUI、HeartbeatReceiver、TaskScheduler、DAGScheduler、MetricsSystem、EventLoggingListener、ExecutorAllocationManager、ContextCleaner 等。其中重点介绍了基于 Jetty 构建的嵌入式 Web 服务 SparkUI 和 ContextCleaner，其他组件将放在后续章节中介绍。

最后还介绍了如何指定用户自定义的 SparkListener、如何指定用户自定义的 Jar 包及文件、投递环境更新事件及其他收尾工作。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用



Chapter 3

第 5 章 Spark 执行环境

“其令郡国各修文学，县满五百户置校官，选其乡之俊造而教学之，庶几先王之道不废，而有以益于天下。”

——《建学令》



本章导读

正如曹操在《建学令》中所言，优秀人才的培育离不开国家政策这个大环境。在人类生活中，总是会依赖于外部的各种环境。一个人学习成绩的好坏，很大的因素取决于他所处的教育环境；一个人健康与否，依赖于他所处的医疗卫生环境；一个人的沟通能力，与自己的家庭环境有很重要的关系。如果某一天你发现自己联系不到亲人、朋友，甚至周围的环境都是陌生的，那么你立刻会处于崩溃的边缘。

使用过 Java 语言的人，应该清楚地记得自己第一次是如何磕磕绊绊老半天，才配置好 Java 环境的过程。有了这次糟糕的体验，我相信每个人在之后配置 Java 环境变量，甚至是 Maven、Scala、SBT 等其他环境变量时，就会变得游刃有余。无论是 Java 程序还是 Scala 程序，都需要运行在其所依托的环境下，脱离了这个环境，你将陷入困境。

Spark 对任务的计算都依托于 Executor 的能力，所有的 Executor 都有自己的 Spark 执行环境 SparkEnv。有了 SparkEnv，就可以将数据存储在存储体系中；就能利用计算引擎对计算任务进行处理，就可以在节点间进行通信等。SparkEnv 还提供了多种多样的内部组件，实现不同的功能。SparkEnv 是一个很重要的组件，虽然在创建 SparkContext 的时候也涉及它（只是因为 local 模式的需要），但是它与 Executor 的关系则更为紧密，所以专门利用一章的内容来详细介绍它。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

本章主要讲解的内容如下。

- 安全管理器 SecurityManager。
- RPC 环境。
- 序列化管理器 SerializerManager。
- 广播管理器 BroadcastManager。
- map 任务输出跟踪器。
- 度量系统。

5.1　SparkEnv 概述

SparkEnv 的私有方法 create 用于创建 SparkEnv，由于 create 方法涉及很多 SparkEnv 内部组件的实例化过程，所以首先通过图 5-1 来了解 SparkEnv 的组成。

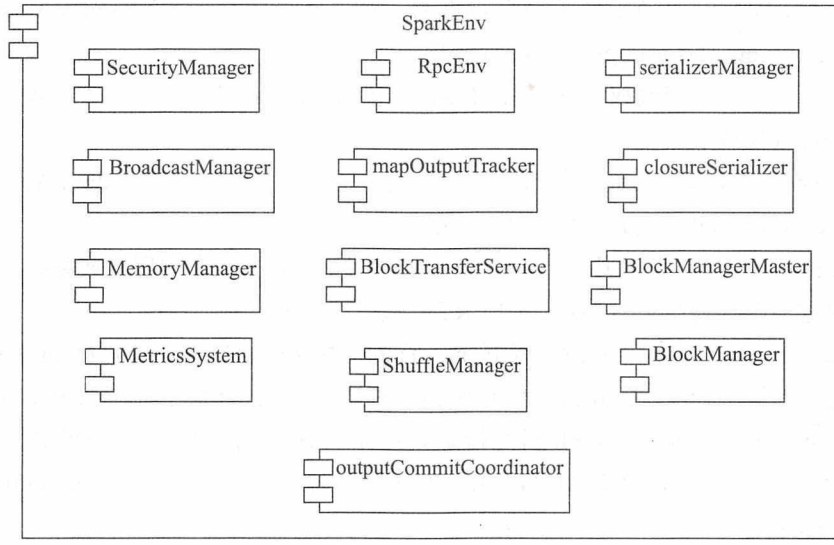


图 5-1　SparkEnv 的组成

有了对 SparkEnv 组成的直观感受，下面我们来对其内部组件逐一进行介绍。

5.2　安全管理器 SecurityManager

SecurityManager 主要对账号、权限及身份认证进行设置和管理。如果 Spark 的部署模式为 YARN，则需要生成 secret key（密钥）并存入 Hadoop UGI。而在其他模式下，则需要设置环境变量 _SPARK_AUTH_SECRET（优先级更高）或 spark.authenticate.secret 属性指定 secret key（密钥）。SecurityManager 还会给当前系统设置默认的口令认证实例。在

仅供非商业用途或交流学习使用

SparkEnv 中创建 SecurityManager 的代码如下。

```
val securityManager = new SecurityManager(conf, ioEncryptionKey)
```

SecurityManager 内部有很多属性，我们先来对这些属性进行介绍。

- authOn：是否开启认证。可以通过 spark.authenticate 属性配置，默认为 false。
- aclsOn：是否对账号进行授权检查。可通过 spark.acls.enable（优先级较高）或 spark.ui.acls.enable（此属性是为了向前兼容）属性进行配置。aclsOn 的默认值为 false。
- adminAcls：管理员账号集合。可以通过 spark.admin.acls 属性配置，默认为空。
- adminAclsGroups：管理员账号所在组的集合。可以通过 spark.admin.acls.groups 属性配置，默认为空。
- viewAcls：有查看权限的账号的集合。包括 adminAcls、defaultAclUsers 及 spark.ui.view.acls 属性配置的用户。
- viewAclsGroups：拥有查看权限的账号，所在组的集合。包括 adminAclsGroups 和 spark.ui.view.acls.groups 属性配置的用户。
- modifyAcls：有修改权限的账号的集合。包括 adminAcls、defaultAclUsers 及 spark.modify.acls 属性配置的用户。
- modifyAclsGroups：拥有修改权限的账号所在组的集合。包括 adminAclsGroups 和 spark.modify.acls.groups 属性配置的用户。
- defaultAclUsers：默认用户。包括系统属性 user.name 指定的用户或系统登录用户或者通过系统环境变量 SPARK_USER 进行设置的用户。
- secretKey：密钥。在 YARN 模式下，首先使用 sparkCookie 从 Hadoop UGI 中获取密钥。如果 Hadoop UGI 没有保存密钥，则生成新的密钥（密钥长度可以通过 spark.authenticate.secretBitLength 属性指定）并存入 Hadoop UGI。其他模式下，则需要设置环境变量 _SPARK_AUTH_SECRET（优先级更高）或 spark.authenticate.secret 属性指定。
- 此外还有很多 SSL 相关的配置。有关 SSL 的内容不属于本书要阐述的范围，感兴趣的读者可以查阅相关资料。



SecurityManager 中设置了默认的口令认证实例 Authenticator，此实例采用匿名内部类实现，用于每次使用 HTTP client 从 HTTP 服务器获取用户的用户名和密码。这是由于 Spark 的节点间通信往往需要动态协商用户名、密码，这种方式灵活地支持了这种需求。设置默认身份认证器的代码如下。

```
if (authOn) {
  // 设置默认的口令认证实例Authenticator，它的getPasswordAuthentication方法用于获取用户
    名、密码
  Authenticator.setDefault(
    new Authenticator() {
```

仅供非商业用途或交流学习使用


```
        override def getPasswordAuthentication(): PasswordAuthentication = {
          var passAuth: PasswordAuthentication = null
          val userInfo = getRequestingURL().getUserInfo()
          if (userInfo != null) {
            val  parts = userInfo.split(":", 2)
            passAuth = new PasswordAuthentication(parts(0), parts(1).toCharArray())
          }
          return passAuth
        }
      }
    )
  }
```

5.3　RPC 环境

RpcEnv 是 Spark 2.x.x 版本中新出现的组件，那么它是不是给 Spark 带来了新的能力呢？对于有些组件是这样的，但是 RpcEnv 组件肩负着另一项历史使命——替代 Spark 2.x.x 以前版本中采用的 Akka。Spark 移除了对 Akka 的使用，这令我感到困惑，“Akka 是一款很不错的分布式消息系统，为什么要替代？正如 3.2 节所说的——可以让用户使用任何版本的 Akka 来编程”。Akka 具有分布式集群下的消息发送、远端同步调用、远端异步调用、路由、持久化、监管、At Least Once Delivery（至少投递一次）等能力，一旦 Akka 被替代，那就意味着 RpcEnv 必须也能支持这些机制。



现在我们来看看 SparkEnv 中的 RpcEnv 是如何被创建的，SparkEnv 中创建 RpcEnv 的代码如下。

```
val systemName = if (isDriver) driverSystemName else executorSystemName
val rpcEnv = RpcEnv.create(systemName, bindAddress, advertiseAddress, port, conf,
  securityManager, clientMode = !isDriver)
```

这段代码首先生成系统名称 systemName，如果当前应用为 Driver（即 SparkEnv 位于 Driver 中），那么 systemName 为 sparkDriver，否则（即 SparkEnv 位于 Executor 中），systemName 为 sparkExecutor。然后调用 RpcEnv 的 create 方法创建 RpcEnv。create 方法中只有两条语句。

```
  val config = RpcEnvConfig(conf, name, bindAddress, advertiseAddress, port,
     securityManager,
    clientMode) // RpcEnvConfig中保存了RpcEnv的配置信息
  new NettyRpcEnvFactory().create(config) // 创建RpcEnv
```

这里的 RpcEnvConfig 实际是一个样例类，用于保存 RpcEnv 的配置信息。实际创建 RpcEnv 的动作发生在 NettyRpcEnvFactory 的 create 方法中，如代码清单 5-1 所示。

代码清单5-1　创建NettyRpcEnv

```
def create(config: RpcEnvConfig): RpcEnv = {
  val sparkConf = config.conf
  val javaSerializerInstance =
```

仅供非商业用途或交流学习使用


```
    new JavaSerializer(sparkConf).newInstance().asInstanceOf[JavaSerializerInstance]
  val nettyEnv =
    new NettyRpcEnv(sparkConf, javaSerializerInstance, config.advertiseAddress,
      config.securityManager)
  if (!config.clientMode) {
    val startNettyRpcEnv: Int => (NettyRpcEnv, Int) = { actualPort =>
      nettyEnv.startServer(config.bindAddress, actualPort)
      (nettyEnv, nettyEnv.address.port)
    }
    try {
      Utils.startServiceOnPort(config.port, startNettyRpcEnv, sparkConf, config.
          name)._1
    } catch {
      case NonFatal(e) =>
        nettyEnv.shutdown()
        throw e
    }
  }
  nettyEnv
}
}
```

代码清单 5-1 中创建 NettyRpcEnv 的步骤如下。

1）创建 javaSerializerInstance。此实例将用于 RPC 传输对象的序列化。

2）创建 NettyRpcEnv。创建 NettyRpcEnv 其实就是对内部各个子组件 TransportConf、Dispatcher、TransportContext、TransportClientFactory、TransportServer 的实例化过程，这些内容都将在后面一一介绍。



3）启动 NettyRpcEnv。这一步首先定义了一个偏函数 startNettyRpcEnv，其函数实际为执行 NettyRpcEnv 的 startServer 方法（startServer 方法的具体内容将在 5.3.7 节进行介绍），最后在启动 NettyRpcEnv 之后返回 NettyRpcEnv 及服务最终使用的端口。这里使用了 Utils 的 startServiceOnPort 方法，startServiceOnPort 实际上是调用了作为参数的偏函数 startNettyRpcEnv，有关 startServiceOnPort 的具体介绍可以参阅附录 A 中的内容。

由于抽象类 RpcEnv 只有一个实现子类 NettyRpcEnv，所以本章不打算展示 RpcEnv 的接口定义，而是直接从介绍 NettyRpcEnv 入手。在正式介绍 NettyRpcEnv 的构造过程之前，我们需要先做一些准备。RpcEndpoint 和 RpcEndpointRef 都是 NettyRpcEnv 中的重要概念，我们需要首先理解它们，再来看 NettyRpcEnv 的构造过程。

5.3.1 RPC 端点 RpcEndpoint

RPC 端点是对 Spark 的 RPC 通信实体的统一抽象，所有运行于 RPC 框架之上的实体都应该继承 RpcEndpoint。Spark 早期版本节点间的消息通信主要采用 Akka 的 Actor，从 Spark 2.0.0 版本开始移除了对 Akka 的依赖，这就意味着 Spark 需要 Actor 的替代品，RPC 端点 RpcEndpoint 由此而生。RpcEndpoint 是对能够处理 RPC 请求，给某一特定服务提供

仅供非商业用途或交流学习使用

本地调用及跨节点调用的 RPC 组件的抽象。

1. RPC 端点 RpcEndpoint 的定义

特质 RpcEndpoint 的定义如代码清单 5-2 所示。

代码清单5-2　RPC端点定义

```
private[spark] trait RpcEndpoint {
  val rpcEnv: RpcEnv
  final def self: RpcEndpointRef = {
    require(rpcEnv != null, "rpcEnv has not been initialized")
    rpcEnv.endpointRef(this)
  }
  def receive: PartialFunction[Any, Unit] = {
    case _ => throw new SparkException(self + " does not implement 'receive'")
  }
  def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
    case _ => context.sendFailure(new SparkException(self + " won't reply
        anything"))
  }
  def onError(cause: Throwable): Unit = {
    throw cause
  }
  def onConnected(remoteAddress: RpcAddress): Unit = {
  }
  def onDisconnected(remoteAddress: RpcAddress): Unit = {
  }
  def onNetworkError(cause: Throwable, remoteAddress: RpcAddress): Unit = {
  }
  def onStart(): Unit = {
  }
  def onStop(): Unit = {
  }
  final def stop(): Unit = {
    val _self = self
    if (_self != null) {
      rpcEnv.stop(_self)
    }
  }
}
```

这里对代码清单 5-2 中所展示的 RpcEndpoint 的各个成员进行介绍。

- rpcEnv：当前 RpcEndpoint 所属的 RpcEnv。
- self：获取 RpcEndpoint 相关联的 RpcEndpointRef。从代码实现看到，其实现实际调用了 RpcEnv 的 endpointRef 方法。由于 RpcEnv 并未实现此方法，所以需要 RpcEnv 的子类来实现。以 NettyRpcEnv 为例，其 endpointRef 方法的实现如代码清单 5-40 所示。
- receive：接收消息并处理，但不需要给客户端回复。
- receiveAndReply：接收消息并处理，需要给客户端回复。回复是通过 RpcCallContext 来实现的。

仅供非商业用途或交流学习使用

- onError：当处理消息发生异常时调用，可以对异常进行一些处理。
- onConnected：当客户端与当前节点连接上之后调用，可以针对连接进行一些处理。
- onDisconnected：当客户端与当前节点的连接断开之后调用，可以针对断开连接进行一些处理。
- onNetworkError：当客户端与当前节点之间的连接发生网络错误时调用，可以针对连接发生的网络错误进行一些处理。
- onStart：在 RpcEndpoint 开始处理消息之前调用，可以在 RpcEndpoint 正式工作之前做一些准备工作。
- onStop：在停止 RpcEndpoint 时调用，可以在 RpcEndpoint 停止的时候做一些收尾工作。
- stop：用于停止当前 RpcEndpoint。从代码实现看到，其实现实际调用了 RpcEnv 的 stop 方法。由于 RpcEnv 并未实现此方法，所以需要其子类来实现。以 NettyRpcEnv 为例，其 stop 方法的实现如 5.3.9 节中代码清单 5-45 所示。

对 Akka 的 Actor 编程模型熟悉的读者，一定觉得 RpcEndpoint 中的一些接口非常类似于 Actor，笔者认为 Spark 2.0.0 版本在去掉了 Akka 的同时，自身实现 RpcEnv 时肯定参考了 Actor 的一些思想。

2. 特质 RpcEndpoint 的继承体系

由于 RpcEndpoint 只是一个特质，除了对接口的定义，并没有任何实现逻辑，所以我们需要看看有哪些子类实现了 RpcEndpoint。RpcEndpoint 的继承体系如图 5-2 所示。

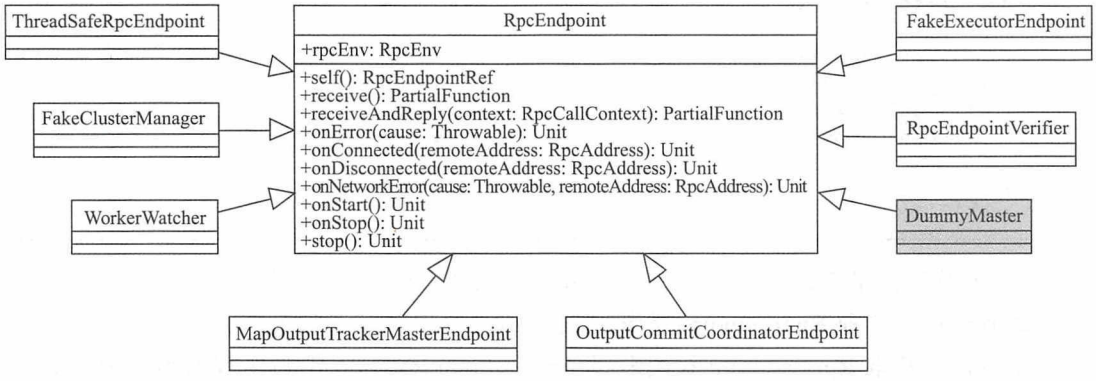


图 5-2 RpcEndpoint 的继承体系

图 5-2 中灰色的子类型 DummyMaster（Dummy 意为虚拟的、假的或傀儡的）正如其名字一样，不是 NettyRpcEnv 中具有真正用途的 RpcEndpoint，而只用于测试。图中显示的 RpcEndpoint 的子类，除了 ThreadSafeRpcEndpoint 之外，都将在具体遇到时再作详细分析。ThreadSafeRpcEndpoint 是继承自 RpcEndpoint 的特质，主要用于对消息的处理，必须是线

仅供非商业用途或交流学习使用

程安全的场景。ThreadSafeRpcEndpoint 对消息的处理都是串行的，即前一条消息处理完才能接着处理下一条消息。ThreadSafeRpcEndpoint 的继承体系如图 5-3 所示。

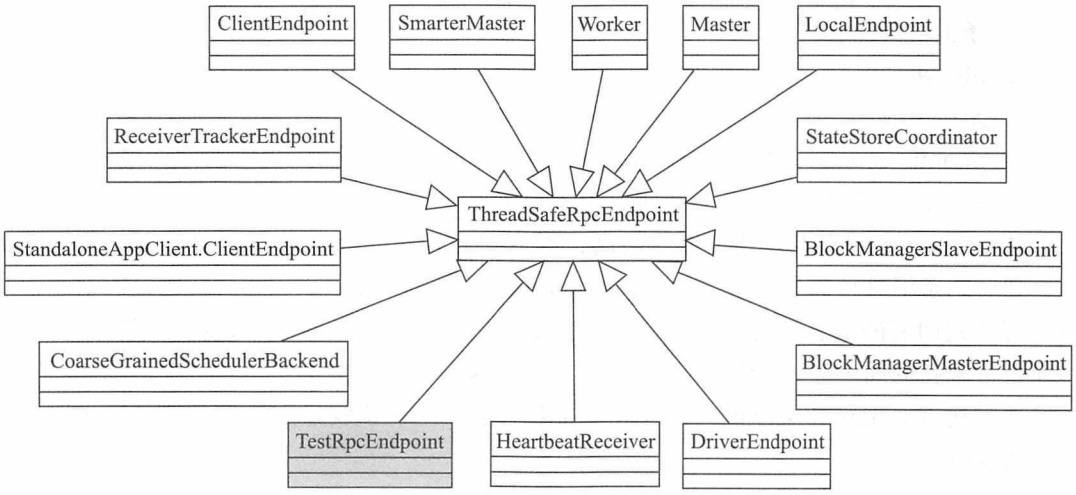


图 5-3　ThreadSafeRpcEndpoint 的继承体系

图 5-3 中的 TestRpcEndpoint 用于测试，其余实现类都在 NettyRpcEnv 中发挥着各自的作用，这些 RpcEndpoint 将在本书后续的各个章节中需要时再作具体介绍。



5.3.2　RPC 端点引用 RpcEndpointRef

如果说 RpcEndpoint 是 Akka 中 Actor 的替代产物，那么 RpcEndpointRef 就是 ActorRef 的替代产物。在 Akka 中只要你持有了一个 Actor 的引用 ActorRef，那么你就可以使用此 ActorRef 向远端的 Actor 发起请求。RpcEndpointRef 也具有同等的效用，要向一个远端的 RpcEndpoint 发起请求，你就必须持有这个 RpcEndpoint 的 RpcEndpointRef。RpcEndpoint 与 RpcEndpointRef 之间的关系如图 5-4 所示。

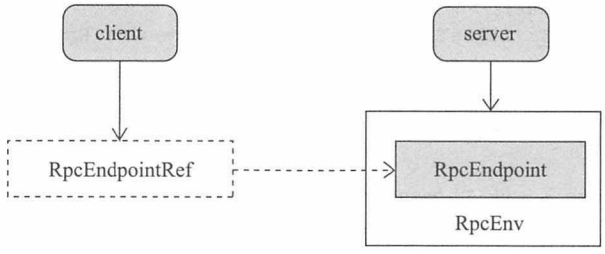


图 5-4　RpcEndpoint 与 RpcEndpointRef 的关系

在正式介绍 RpcEndpointRef 之前，先要理解什么是消息投递规则。

仅供非商业用途或交流学习使用

1. 消息投递规则

一般而言，消息投递有下面 3 种情况。

- at-most-once：意味着每条应用了这种机制的消息会被投递 0 次或 1 次。可以说这条消息可能会丢失。
- at-least-once：意味着每条应用了这种机制的消息潜在地存在多次投递尝试并保证至少会成功一次。就是说这条消息可能会重复但是不会丢失。
- exactly-once：意味着每条应用了这种机制的消息只会向接收者准确地发送一次。换言之，这种消息既不会丢失，也不会重复。

at-most-once 的成本最低且性能最高，因为它在发送完消息后不会尝试去记录任何状态，然后这条消息将被抛之脑后。at-least-once 需要发送者必须认识它所发送过的消息，并对没有收到回复的消息进行发送重试。这就要求接收者对消息的处理必须是幂等的，否则可能会带来新的问题。exactly-once 的成本是三者中最高，性能却又是三者中最差的。它除了要求发送者有记忆和重试能力，还要求接收者能够认识接收过的消息并能过滤出那些重复的消息投递。

2. RPC 端点引用 RpcEndpointRef 的定义

有了对消息投递规则的理解，我们现在开始介绍 RpcEndpointRef。抽象类 RpcEndpointRef 定义了所有 RpcEndpoint 引用的属性与接口，如代码清单 5-3 所示。



代码清单5-3 RpcEndpointRef的接口定义

```
private[spark] abstract class RpcEndpointRef(conf: SparkConf)
  extends Serializable with Logging {
  private[this] val maxRetries = RpcUtils.numRetries(conf)
  private[this] val retryWaitMs = RpcUtils.retryWaitMs(conf)
  private[this] val defaultAskTimeout = RpcUtils.askRpcTimeout(conf)
  def address: RpcAddress
  def name: String
  def send(message: Any): Unit
  def ask[T: ClassTag](message: Any, timeout: RpcTimeout): Future[T]
  def ask[T: ClassTag](message: Any): Future[T] = ask(message, defaultAskTimeout)
  def askWithRetry[T: ClassTag](message: Any): T = askWithRetry(message,
     defaultAskTimeout)
  def askWithRetry[T: ClassTag](message: Any, timeout: RpcTimeout): T = {
    var attempts = 0
    var lastException: Exception = null
    while (attempts < maxRetries) {
      attempts += 1
      try {
        val future = ask[T](message, timeout)
        val result = timeout.awaitResult(future)
        if (result == null) {
          throw new SparkException("RpcEndpoint returned null")
        }
        return result
      } catch {
```

仅供非商业用途或交流学习使用


```
          case ie: InterruptedException => throw ie
        case e: Exception =>
            lastException = e
            logWarning(s"Error sending message [message = $message] in $attempts
                attempts", e)
        }

        if (attempts < maxRetries) {
          Thread.sleep(retryWaitMs)
        }
      }

      throw new SparkException(
        s"Error sending message [message = $message]", lastException)
    }

  }
```

从代码清单 5-3 中看到，抽象类 RpcEndpointRef 中有 3 个属性，分别如下。

- maxRetries：RPC 最大重新连接次数。可以使用 spark.rpc.numRetries 属性进行配置，默认为 3 次。
- retryWaitMs：RPC 每次重新连接需要等待的毫秒数。可以使用 spark.rpc.retry.wait 属性进行配置，默认值为 3 秒。
- defaultAskTimeout：RPC 的 ask 操作的默认超时时间。可以使用 spark.rpc.askTimeout 或者 spark.network.timeout 属性进行配置，默认值为 120 秒。spark.rpc.askTimeout 属性的优先级更高。

这三个属性都是通过调用工具类 RpcUtils 的方法得到的，想了解 RpcUtils 的介绍，可以参阅附录 H。

接着来看一看 RpcEndpointRef 中各个方法的功能。

- address：返回当前 RpcEndpointRef 对应 RpcEndpoint 的 RPC 地址（RpcAddress）。
- name：返回当前 RpcEndpointRef 对应 RpcEndpoint 的名称。
- send：发送单向异步的消息。所谓“单向”就是发送完后就会忘记此次发送，不会有任何状态要记录，也不会期望得到服务端的回复。send 采用了 at-most-once 的投递规则。RpcEndpointRef 的 send 方法非常类似于 Akka 中 Actor 的 tell 方法。
- ask[T: ClassTag](message: Any)：以默认的超时时间作为 timeout 参数，调用 ask[T: ClassTag](message: Any, timeout: RpcTimeout) 方法。
- askWithRetry[T: ClassTag](message: Any, timeout: RpcTimeout)：发送同步的请求，此类请求将会被 RpcEndpoint 接收，并在指定的超时时间内等待返回类型为 T 的处理结果。当此方法抛出 SparkException 时，将会进行请求重试，直到超过了默认的重试次数为止。由于此类方法会重试，因此要求服务端对消息的处理是幂等的。此



仅供非商业用途或交流学习使用

方法也采用了 at-least-once 的投递规则。此方法也非常类似于 Akka 中采用了 at-least-once 机制的 Actor 的 ask 方法。

- askWithRetry[T: ClassTag](message: Any)：以默认的超时时间作为 timeout 参数，调用 askWithRetry[T: ClassTag](message: Any, timeout: RpcTimeout)。

小贴士：Akka 的 Actor 模型一般提供的消息都属于 at-most-once，那是因为大多数场景不需要有状态的消息投递，如 Web 服务器请求。当你有强一致性需求时，才应该启用 Akka 的 at-least-once 机制。

5.3.3 创建传输上下文 TransportConf

根据 3.2.1 节的介绍，我们知道 TransportConf 是 RPC 框架中的配置类。由于 RPC 环境 RpcEnv 的底层也依赖于数据总线，因此需要创建传输上下文 TransportConf。创建 TransportConf 是构造 NettyRpcEnv 的过程中的第一步，代码如下。

```
private[netty] val transportConf = SparkTransportConf.fromSparkConf(
  conf.clone.set("spark.rpc.io.numConnectionsPerPeer", "1"),
  "rpc",
  conf.getInt("spark.rpc.io.threads", 0))
```

这里用到了 3.2.1 节介绍的 SparkTransportConf，可以看到在调用 SparkTransportConf 的 fromSparkConf 方法之前，首先对 SparkConf 进行了克隆，然后设置了 spark.rpc.io.numConnectionsPerPeer，此属性的格式在 3.2.2 节分析 TransportClientFactory 的成员变量时介绍过，由于当前模块名为 rpc，所以为 spark.rpc.io.numConnectionsPerPeer。我们还看到这里通过 spark.rpc.io.threads 属性来设置 Netty 传输线程数。



5.3.4 消息调度器 Dispatcher

创建消息调度器 Dispatcher 是有效提高 NettyRpcEnv 对消息异步处理并最大提升并行处理能力的前提。Dispatcher 负责将 RPC 消息路由到要该对此消息处理的 RpcEndpoint（RPC 端点）。创建 Dispatcher 的代码如下。

```
private val dispatcher: Dispatcher = new Dispatcher(this)
```

1. 消息调度器 Dispatcher 的概述

想要把 Dispatcher 解释得足够明白，最好的方式是先弄明白其中涉及的概念。Dispatcher 中的概念包括以下几个。

- RpcEndpoint：RPC 端点，即 RPC 分布式环境中一个具体的实例，其可以对指定的消息进行处理。由于 RpcEndpoint 是一个特质，所以需要提供 RpcEndpoint 的实现类。特质 RpcEndpoint 已在前文详细介绍，此处不再赘述。
- RpcEndpointRef：RPC 端点引用，即 RPC 分布式环境中一个具体实体的引用，所

仅供非商业用途或交流学习使用

谓引用实际是“spark://host:port/name”这种格式的地址。其中，host 为端点所在 RPC 服务所在的主机 IP，port 是端点所在 RPC 服务的端口，name 是端点实例的名称。抽象类 RpcEndpointRef 已在前文详细介绍，此处不再赘述。

- InboxMessage：Inbox 盒子内的消息。InboxMessage 是一个特质，所有类型的 RPC 消息都继承自 InboxMessage。InboxMessage 的继承体系可以用图 5-5 表示。

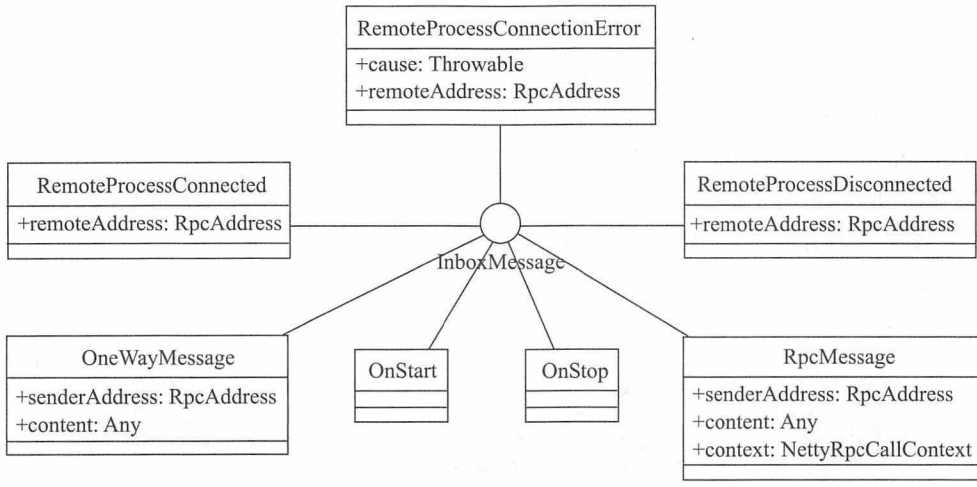


图 5-5　InboxMessage 的继承体系

图 5-5 展示了 InboxMessage 的多种实现类，其作用分别如下。

- OneWayMessage：RpcEndpoint 处理此类型的消息后不需要向客户端回复信息。
- RpcMessage：RPC 消息，RpcEndpoint 处理完此消息后需要向客户端回复信息。
- OnStart：用于 Inbox 实例化后，再通知与此 Inbox 相关联的 RpcEndpoint 启动。
- OnStop：用于 Inbox 停止后，通知与此 Inbox 相关联的 RpcEndpoint 停止。
- RemoteProcessConnected：此消息用于告诉所有的 RpcEndpoint，有远端的进程已经与当前 RPC 服务建立了连接。
- RemoteProcessDisconnected：此消息用于告诉所有的 RpcEndpoint，有远端的进程已经与当前 RPC 服务断开了连接。
- RemoteProcessConnectionError：此消息用于告诉所有的 RpcEndpoint，与远端某个地址之间的连接发生了错误。

- Inbox：端点内的盒子。每个 RpcEndpoint 都有一个对应的盒子，这个盒子里有个存储 InboxMessage 消息的列表 messages。所有的消息将缓存在 messages 列表里面，并由 RpcEndpoint 异步处理这些消息。
- EndpointData：RPC 端点数据，它包括了 RpcEndpoint、NettyRpcEndpointRef 及 Inbox 等属于同一个端点的实例。Inbox 与 RpcEndpoint、NettyRpcEndpointRef 通过

仅供非商业用途或交流学习使用

此 EndpointData 相关联。

有了对以上概念的介绍，现在来看看 Dispatcher 中的一些成员变量。

- endpoints：端点实例名称与端点数据 EndpointData 之间映射关系的缓存。有了这个缓存，就可以使用端点名称从中快速获取或删除 EndpointData 了。
- endpointRefs：端点实例 RpcEndpoint 与端点实例引用 RpcEndpointRef 之间映射关系的缓存。有了这个缓存，就可以使用端点实例从中快速获取或删除端点实例引用了。
- receivers：存储端点数据 EndpointData 的阻塞队列。只有 Inbox 中有消息的 EndpointData 才会被放入此阻塞队列。
- stopped：Dispatcher 是否停止的状态。
- threadpool：用于对消息进行调度的线程池。此线程池运行的任务都是 MessageLoop（将在代码清单 5-4、代码清单 5-5 中详细介绍）。

有了对 Dispatcher 涉及概念及属性的了解，我们就可以更容易理解 Dispatcher 的内存模型了，如图 5-6 所示。

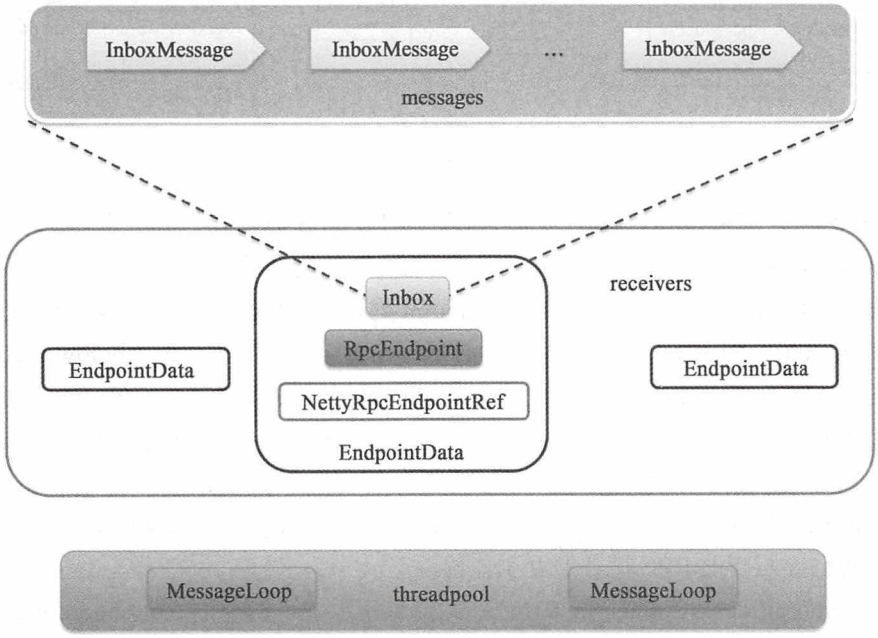


图 5-6 Dispatcher 的内存模型

2. Dispatcher 的调度原理

在创建 Dispatcher 的最后会创建对消息进行调度的线程池 threadpool，如代码清单 5-4 所示。

仅供非商业用途或交流学习使用

代码清单5-4　创建Dispatcher中的线程池

```
private val threadpool: ThreadPoolExecutor = {
  val numThreads = nettyEnv.conf.getInt("spark.rpc.netty.dispatcher.numThreads",
    math.max(2, Runtime.getRuntime.availableProcessors()))
  val pool = ThreadUtils.newDaemonFixedThreadPool(numThreads, "dispatcher-event-
     loop")
  for (i <- 0 until numThreads) {
    pool.execute(new MessageLoop)
  }
  pool
}
```

根据代码清单 5-4，可以看到创建 threadpool 线程池的步骤如下。

1）获取此线程池的大小 numThreads。此线程池的大小默认为 2 与当前系统可用处理器数量之间的最大值，也可以使用 spark.rpc.netty.dispatcher.numThreads 属性配置。

2）创建线程池。此线程池是固定大小的线程池，并且启动的线程都以后台线程方式运行，且线程名以 dispatcher-event-loop 为前缀。

3）启动多个运行 MessageLoop 任务的线程，这些线程的数量与 threadpool 线程池的大小相同。

4）返回此线程池的引用。

MessageLoop 实现了 Java 的 Runnable 接口，其实现如代码清单 5-5 所示。



代码清单5-5　MessageLoop的实现

```
private class MessageLoop extends Runnable {
  override def run(): Unit = {
    try {
      while (true) {
        try {
          val data = receivers.take()
          if (data == PoisonPill) {
            // Put PoisonPill back so that other MessageLoops can see it.
            receivers.offer(PoisonPill)
            return
          }
          data.inbox.process(Dispatcher.this)
        } catch {
          case NonFatal(e) => logError(e.getMessage, e)
        }
      }
    } catch {
      case ie: InterruptedException => // exit
    }
  }
}
```

根据代码清单 5-5，我们看到 MessageLoop 果如其名，在循环过程中不断对新的消息进行处理。每次循环中的逻辑如下。

仅供非商业用途或交流学习使用

1）从 receivers 中获取 EndpointData。receivers 中的 EndpointData，其 Inbox 的 messages 列表中肯定有了新的消息。换言之，只有 Inbox 的 messages 列表中有了新的消息，此 EndpointData 才会被放入 receivers 中。由于 receivers 是个阻塞队列，所以当 receivers 中没有 EndpointData 时，MessageLoop 线程会被阻塞。

2）如果取到的 EndpointData 是“毒药”（PoisonPill），那么此 MessageLoop 线程将退出（通过 return 语句）。这里有个动作就是将 PoisonPill 重新放入到 receivers 中，这是因为 threadpool 线程池极有可能不止一个 MessageLoop 线程，为了让大家都“毒发身亡”，还需要把“毒药”放回到 receivers 中，这样其他“活着”的线程就会再次误食“毒药”，达到所有 MessageLoop 线程都结束的效果。

3）如果取到的 EndpointData 不是“毒药”，那么调用 EndpointData 中 Inbox 的 process 方法对消息进行处理。

上文的 MessageLoop 任务实际是将消息交给 EndpointData 中 Inbox 的 process 方法处理的，但在正式介绍 process 之前，先来看看 Inbox 中的一些成员属性，这样有助于之后对 process 方法的理解。Inbox 中的成员属性如下。

- messages：消息列表。用于缓存需要由对应 RpcEndpoint 处理的消息，即与 Inbox 在同一 EndpointData 中的 RpcEndpoint。Inbox 构造的最后是一些有些古怪的代码，如代码清单 5-6 所示。



代码清单5-6 Inbox向自身的messages列表中放入OnStart消息

```
inbox.synchronized {
  messages.add(OnStart)
}
```

上述代码将向 Inbox 自身的 messages 列表中放入 OnStart 消息，为什么会有这么一段代码？此处先不解释这段代码的具体用途。

- stopped：Inbox 的停止状态。
- enableConcurrent：是否允许多个线程同时处理 messages 中的消息。
- numActiveThreads：激活线程的数量，即正在处理 messages 中消息的线程数量。

介绍完 Inbox 中的成员属性，我们一起来看看其 process 方法的实现，如代码清单 5-7 所示。

代码清单5-7 Inbox中的消息处理

```
def process(dispatcher: Dispatcher): Unit = {
  var message: InboxMessage = null
  inbox.synchronized {
    if (!enableConcurrent && numActiveThreads != 0) {
      return
    }
    message = messages.poll()
    if (message != null) {
```

仅供非商业用途或交流学习使用


```
      numActiveThreads += 1
    } else {
      return
    }
  }
  while (true) {
    safelyCall(endpoint) {
      message match {
        case RpcMessage(_sender, content, context) =>
          try {
            endpoint.receiveAndReply(context).applyOrElse[Any, Unit](content, {
                msg =>
              throw new SparkException(s"Unsupported message $message from ${_
                  sender}")
            })
          } catch {
            case NonFatal(e) =>
              context.sendFailure(e)
              // Throw the exception -- this exception will be caught by the
                  safelyCall function.
              // The endpoint's onError function will be called.
              throw e
          }

        case OneWayMessage(_sender, content) =>
          endpoint.receive.applyOrElse[Any, Unit](content, { msg =>
            throw new SparkException(s"Unsupported message $message from ${_
                sender}")
          })

        case OnStart =>
          endpoint.onStart()
          if (!endpoint.isInstanceOf[ThreadSafeRpcEndpoint]) {
            inbox.synchronized {
              if (!stopped) {
                enableConcurrent = true
              }
            }
          }

        case OnStop =>
          val activeThreads = inbox.synchronized { inbox.numActiveThreads }
          assert(activeThreads == 1,
            s"There should be only a single active thread but found
                $activeThreads threads.")
          dispatcher.removeRpcEndpointRef(endpoint)
          endpoint.onStop()
          assert(isEmpty, "OnStop should be the last message")

        case RemoteProcessConnected(remoteAddress) =>
          endpoint.onConnected(remoteAddress)

        case RemoteProcessDisconnected(remoteAddress) =>
          endpoint.onDisconnected(remoteAddress)

        case RemoteProcessConnectionError(cause, remoteAddress) =>
```



仅供非商业用途或交流学习使用


```
            endpoint.onNetworkError(cause, remoteAddress)
          }
        }

        inbox.synchronized {
          // "enableConcurrent" will be set to false after 'onStop' is called, so we
              should check it
          // every time.
          if (!enableConcurrent && numActiveThreads != 1) {
            // If we are not the only one worker, exit
            numActiveThreads -= 1
            return
          }
          message = messages.poll()
          if (message == null) {
            numActiveThreads -= 1
            return
          }
        }
      }
    }
```

根据代码清单 5-7，Inbox 对消息处理的步骤如下。

1）进行线程并发检查。具体是，如果不允许多个线程同时处理 messages 中的消息（enableConcurrent 为 false），并且当前激活线程数（numActiveThreads）不为 0，这说明已经有线程在处理消息，所以当前线程不允许再去处理消息（使用 return 返回）。



2）从 messages 中获取消息。如果有消息未处理，则当前线程需要处理此消息，因而算是一个新的激活线程（需要将 numActiveThreads 加 1）。如果 messages 中没有消息了（一般发生在多线程情况下），则直接返回。

3）根据消息类型进行匹配，并执行对应的逻辑。这里有个小技巧值得借鉴，那就是匹配执行的过程中也许会发生错误，当发生错误的时候，我们希望当前 Inbox 所对应 RpcEndpoint 的错误处理方法 onError 可以接收到这些错误信息。Inbox 的 safelyCall 方法给我们提供了这方面的实现，如代码清单 5-8 所示。

代码清单5-8　safelyCall的实现

```
private def safelyCall(endpoint: RpcEndpoint)(action: => Unit): Unit = {
  try action catch {
    case NonFatal(e) =>
      try endpoint.onError(e) catch {
        case NonFatal(ee) => logError(s"Ignoring error", ee)
      }
  }
}
```

4）对激活线程数量进行控制。当第 3）步对消息处理完毕后，当前线程作为之前已经激活的线程是否还有存在的必要呢？这里有两个判断：如果不允许多个线程同时处理 messages 中的消息并且当前激活的线程数多于 1 个，那么需要当前线程退出并将

仅供非商业用途或交流学习使用

numActiveThreads 减 1；如果 messages 已经没有消息要处理了，这说明当前线程无论如何也该返回并将 numActiveThreads 减 1。

注意　第 1）步、第 2）步及第 4）步位于 Inbox 的锁保护之下，是因为 messages 是普通的 java.util.LinkedList，LinkedList 本身不是线程安全的，所以为了增加并发安全性，需要通过同步保护。

3. Inbox 的消息来源

MessageLoop 线程的执行逻辑是不断地消费各个 EndpointData 中 Inbox 里的消息，但是 EndpointData 是何时放入 receivers 中的？ Inbox 里的消息来自哪里？ Dispatcher 中有很多完成这些功能的方法，让我们一起看看 Dispatcher 中与此相关的一些方法。

（1）注册 RpcEndpoint

Dispatcher 的 registerRpcEndpoint 方法用于注册 RpcEndpoint，这个方法的副作用便可以将 EndpointData 放入 receivers，其实现如代码清单 5-9 所示。

代码清单5-9　注册RpcEndpoint

```
def registerRpcEndpoint(name: String, endpoint: RpcEndpoint): NettyRpcEndpointRef = {
  val addr = RpcEndpointAddress(nettyEnv.address, name)
  val endpointRef = new NettyRpcEndpointRef(nettyEnv.conf, addr, nettyEnv)
  synchronized {
    if (stopped) {
      throw new IllegalStateException("RpcEnv has been stopped")
    }
    if (endpoints.putIfAbsent(name, new EndpointData(name, endpoint,
        endpointRef)) != null) {
      throw new IllegalArgumentException(s"There is already an RpcEndpoint called
          $name")
    }
    val data = endpoints.get(name)
    endpointRefs.put(data.endpoint, data.ref)
    receivers.offer(data)  // for the OnStart message
  }
  endpointRef
}
```

从代码清单 5-9 可以看到，注册 RpcEndpoint 的步骤如下。

1）使用当前 RpcEndpoint 所在 NettyRpcEnv 的地址和 RpcEndpoint 的名称创建 RpcEndpointAddress 对象。

2）创建 RpcEndpoint 的引用对象——NettyRpcEndpointRef。

3）创建 EndpointData，并放入 endpoints 缓存。

4）将 RpcEndpoint 与 NettyRpcEndpointRef 的映射关系放入 endpointRefs 缓存。

5）将 EndpointData 放入阻塞队列 receivers 的队尾。MessageLoop 线程异步获取到此 EndpointData，并处理其 Inbox 中刚刚放入的 OnStart 消息，最终调用 RpcEndpoint 的

仅供非商业用途或交流学习使用

OnStart 方法在 RpcEndpoint 开始处理消息之前做一些准备工作。

小贴士：由于 EndpointData 是新创建的，所以在构造 EndpointData 的过程中也会构造它内部的 Inbox。因此 Inbox 在构造的最后会执行代码清单 5-6 所示的逻辑，向 Inbox 自身的 messages 列表放入 OnStart 消息。随后在注册 RpcEndpoint 的第 5）步，当前 Inbox 所属的 EndpointData 就被放入 receivers 中。最后 MessageLoop 线程就会取出此 EndpointData 处理，进而调用当前 Inbox 的 process 方法处理 OnStart。Inbox 的 process 方法处理 OnStart 的效果是在 Inbox 自身初始化完毕后，再启动与此 Inbox 相关联的 RpcEndpoint（可以参见代码清单 5-7 中对 OnStart 消息匹配后的执行过程）。

6）返回 NettyRpcEndpointRef。

（2）对 RpcEndpoint 去注册

Dispatcher 的 stop 方法（见代码清单 5-10）用于对 RpcEndpoint 的注册。

代码清单5-10 Dispatcher的stop方法

```
def stop(rpcEndpointRef: RpcEndpointRef): Unit = {
  synchronized {
    if (stopped) {
      return
    }
    unregisterRpcEndpoint(rpcEndpointRef.name)
  }
}
```



读者可能对 stop 这个方法的命名有些怀疑，不应该是停止 Dispatcher 的吗？阅读了代码清单 5-10 中 stop 方法的实现后，你就会释然。代码清单 5-10 中 stop 方法的执行步骤为：首先判断 Dispatcher 是否已经停止，如果 Dispatcher 未停止，则调用 Dispatcher 的 unregisterRpcEndpoint 方法对 RpcEndpoint 去注册。

Dispatcher 的 unregisterRpcEndpoint 方法用于对 RpcEndpoint 去注册，这个方法的副作用也可以将 EndpointData 放入 receivers，其实现如代码清单 5-11 所示。

代码清单5-11 对RpcEndpoint去注册

```
private def unregisterRpcEndpoint(name: String): Unit = {
  val data = endpoints.remove(name)
  if (data != null) {
    data.inbox.stop()
    receivers.offer(data)  // for the OnStop message
  }
}
```

从代码清单 5-11 可以看到，去注册的步骤如下。

1）从 endpoints 中移除 EndpointData。

2）调用 EndpointData 中 Inbox 的 stop 方法停止 Inbox。

仅供非商业用途或交流学习使用

3）将 EndpointData 重新放入 receivers 中。

代码本身很简单，可是奇怪的是为什么 EndpointData 从 endpoints 中移除后，最后还要放入 receivers？ EndpointData 虽然移除了，但是对应的 RpcEndpointRef 并没有从 endpointRefs 缓存中移除，这又是为什么？这里先不直接回答这两个问题。

当要移除一个 EndpointData 时，其 Inbox 可能正在对消息进行处理，所以不能贸然停止。这里采用了更平滑的停止方式，即调用了 Inbox 的 stop 方法来平滑过渡，stop 方法的实现如代码清单 5-12 所示。

代码清单5-12　Inbox的stop方法

```
def stop(): Unit = inbox.synchronized {
  if (!stopped) {
    enableConcurrent = false
    stopped = true
    messages.add(OnStop)
  }
}
```

Inbox 的 stop 方法的处理步骤如下。

1）根据之前的分析，MessageLoop 有“允许并发运行”和“不允许并发运行”两种情况。对于允许并发的情况，为了确保安全，应该将 enableConcurrent 设置为 false。

2）设置当前 Inbox 为停止状态。

3）向 messages 中添加 OnStop 消息。根据代码清单 5-5 中对 MessageLoop 的分析，为了能够处理 OnStop 消息，只有 Inbox 所属的 EndpointData 放入 receivers 中，其 messages 列表中的消息才会被处理，这回答了刚才提出的第一个问题。为了实现平滑停止，OnStop 消息最终将匹配代码清单 5-7，调用 Dispatcher 的 removeRpcEndpointRef 方法，将 RpcEndpoint 与 RpcEndpointRef 的映射从缓存 endpointRefs 中移除，这回答了刚才的第二个问题。在匹配执行 OnStop 消息的最后，将调用 RpcEndpoint 的 onStop 方法对 RpcEndpoint 停止。

（3）将消息提交给指定的 RpcEndpoint

Dispatcher 的 postMessage 用于将消息提交给指定的 RpcEndpoint，其实现如代码清单 5-13 所示。

代码清单5-13　postMessage的实现

```
private def postMessage(
    endpointName: String,
    message: InboxMessage,
    callbackIfStopped: (Exception) => Unit): Unit = {
  val error = synchronized {
    val data = endpoints.get(endpointName)
    if (stopped) {
      Some(new RpcEnvStoppedException())
    } else if (data == null) {
      Some(new SparkException(s"Could not find $endpointName."))
    } else {
```

仅供非商业用途或交流学习使用


```
        data.inbox.post(message)
        receivers.offer(data)
        None
      }
    }
    // We don't need to call 'onStop' in the 'synchronized' block
    error.foreach(callbackIfStopped)
  }
```

从代码清单 5-13 可以看出，postMessage 方法的处理步骤如下。

1）根据端点名称 endpointName 从缓存 endpoints 中获取 EndpointData。

2）如果当前 Dispatcher 没有停止并且缓存 endpoints 中确实存在名为 endpointName 的 EndpointData，那么将调用 EndpointData 对应 Inbox 的 post 方法将消息加入 Inbox 的消息列表中，因此还需要将 EndpointData 推入 receivers，以便 MessageLoop 处理此 Inbox 中的消息。Inbox 的 post 方法的实现如代码清单 5-14 所示，其逻辑为 Inbox 未停止时向 messages 列表加入消息。

代码清单5-14 Inbox的post方法

```
def post(message: InboxMessage): Unit = inbox.synchronized {
  if (stopped) {
    // We already put "OnStop" into "messages", so we should drop further
        messages
    onDrop(message)
  } else {
    messages.add(message)
    false
  }
}
```



此外，Dispatcher 中还有一些方法间接使用了 Dispatcher 的 postMessage 方法，如代码清单 5-15 所示。

代码清单5-15 对postMessage方法的间接使用

```
def postToAll(message: InboxMessage): Unit = {
  val iter = endpoints.keySet().iterator()
  while (iter.hasNext) {
    val name = iter.next
    postMessage(name, message, (e) => logWarning(s"Message $message dropped. ${e.
        getMessage}"))
  }
}

/** Posts a message sent by a remote endpoint. */
def postRemoteMessage(message: RequestMessage, callback: RpcResponseCallback):
   Unit = {
  val rpcCallContext =
    new RemoteNettyRpcCallContext(nettyEnv, callback, message.senderAddress)
  val rpcMessage = RpcMessage(message.senderAddress, message.content, rpcCallContext)
  postMessage(message.receiver.name, rpcMessage, (e) => callback.onFailure(e))
```

仅供非商业用途或交流学习使用


```
}

/** Posts a message sent by a local endpoint. */
def postLocalMessage(message: RequestMessage, p: Promise[Any]): Unit = {
  val rpcCallContext =
    new LocalNettyRpcCallContext(message.senderAddress, p)
  val rpcMessage = RpcMessage(message.senderAddress, message.content, rpcCallContext)
  postMessage(message.receiver.name, rpcMessage, (e) => p.tryFailure(e))
}

/** Posts a one-way message. */
def postOneWayMessage(message: RequestMessage): Unit = {
  postMessage(message.receiver.name, OneWayMessage(message.senderAddress, message.
      content),
    (e) => throw e)
}
```

代码清单 5-15 中的方法除了 postOneWayMessage 方法，其他需要回复的消息中都封装了 RpcCallContext。RpcCallContext 是用于回调客户端的上下文，其定义如代码清单 5-16 所示。

代码清单5-16　RpcCallContext的定义

```
private[spark] trait RpcCallContext {
  def reply(response: Any): Unit
  def sendFailure(e: Throwable): Unit
  def senderAddress: RpcAddress
}
```



代码清单 5-16 中的 RpcCallContext 一共定义了三个接口。

- reply：用于向发送者回复信息。
- sendFailure：用于向发送者发送失败信息。
- senderAddress：用于获取发送者的地址。

特质 RpcCallContext 的继承体系如图 5-7 所示。

（4）停止 Dispatcher

Dispatcher 的 stop 方法用来停止 Dispatcher，其实现如代码清单 5-17 所示。

代码清单5-17　停止Dispatcher

```
def stop(): Unit = {
  synchronized {
    if (stopped) {
      return
    }
    stopped = true
  }
  // Stop all endpoints. This will queue all endpoints for processing by the
      message loops.
  endpoints.keySet().asScala.foreach(unregisterRpcEndpoint)
  // Enqueue a message that tells the message loops to stop.
  receivers.offer(PoisonPill)
  threadpool.shutdown()
```

仅供非商业用途或交流学习使用


```
}
```

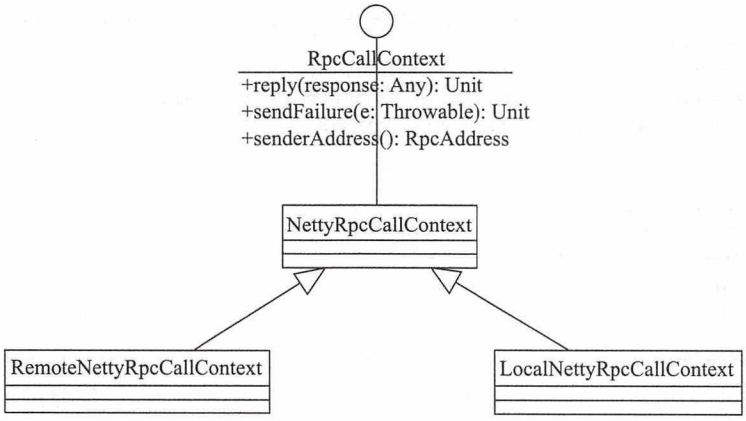


图 5-7 RpcCallContext 的继承体系

代码清单 5-17 中 stop 方法的执行步骤如下。

1）如果 Dispatcher 还未停止，则将自身状态修改为已停止。

2）对 endpoints 中的所有 EndpointData 去注册。这里通过调用 unregisterRpcEndpoint 方法（见代码清单 5-11），将向 endpoints 中的每个 EndpointData 的 Inbox 里放置 OnStop 消息。

3）向 receivers 中投放“毒药”。

由于 Dispatcher 都停止了，所以 Dispatcher 中的所有 MessageLoop 线程也就没有存在的必要了。

4）关闭 threadpool 线程池。

经过对 Dispatcher 和 MessageLoop 的分析，我们现在可以对图 5-6 中展现的内容进行扩展，增加一些运行时的执行流程，如图 5-8 所示。

图 5-8 中的序号代表其中的简要步骤，具体解释如下。

- 序号①表示调用 Inbox 的 post 方法将消息放入 messages 列表中。
- 序号②表示将有消息的 Inbox 相关联的 EndpointData 放入 receivers。
- 序号③表示 MessageLoop 每次循环首先从 receivers 中获取 EndpointData。
- 序号④表示执行 EndpointData 中 Inbox 的 process 方法对消息进行具体处理。

5.3.5 创建传输上下文 TransportContext

创建传输上下文 TransportContext 是 NettyRpcEnv 提供服务端与客户端能力的前提。创建 TransportContext 的代码如下。

仅供非商业用途或交流学习使用

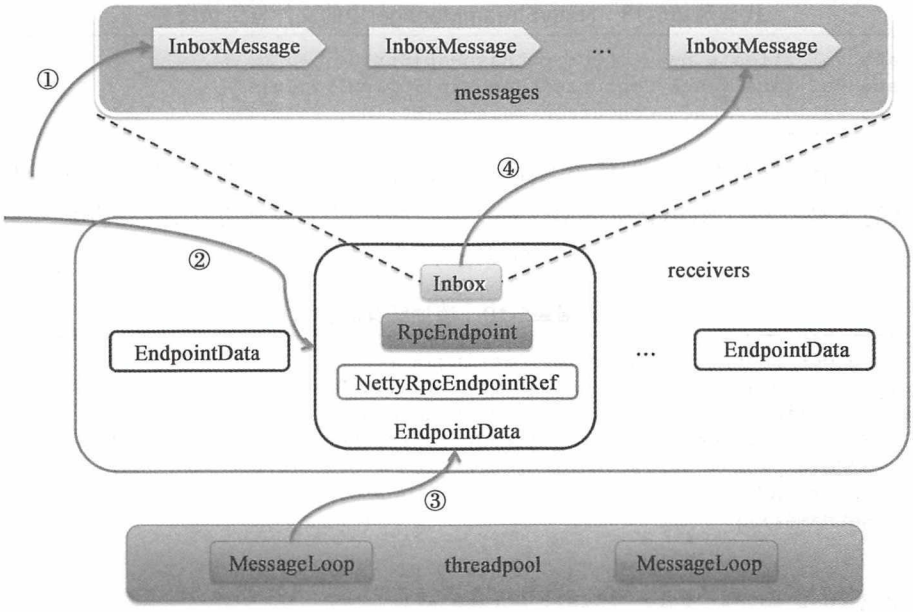


图 5-8　Dispatcher 中 MessageLoop 线程的执行流程

```
private val streamManager = new NettyStreamManager(this)
private val transportContext = new TransportContext(transportConf,
  new NettyRpcHandler(dispatcher, this, streamManager))
```

关于 TransportContext，在 3.2 节已经有足够多的介绍，所以这里重点来看看其构造器中传入的 RpcHandler 参数。这里用于构造 TransportContext 的 RpcHandler 实际是其实现类 NettyRpcHandler，NettyRpcHandler 的构造器里则以刚刚新建的 NettyStreamManager 实例作为参数。本节将首先介绍 NettyStreamManager，然后来看看 NettyRpcHandler 的实现。

1. NettyStreamManager 详解

NettyStreamManager 实现了 StreamManager，专用于为 NettyRpcEnv 提供文件服务的能力。创建 StreamManager 的代码如下。

```
private val streamManager = new NettyStreamManager(this)
```

从上面代码可以看出，此处创建的 streamManager 的实际类型为 NettyStreamManager。NettyStreamManager 用于提供 NettyRpcEnv 的文件流服务，这替代了在 Spark 1.x.x 版本中使用 Jetty 实现的 HttpFileServer。在 3.2.6 节的代码清单 3-27 中我们曾经介绍过，RpcHandler 的 getStreamManager 方法返回的类型为 StreamManager。NettyStreamManager 正是继承了 StreamManager。NettyStreamManager 中定义了三个文件与目录缓存，如代码清单 5-18 所示。

仅供非商业用途或交流学习使用

代码清单5-18 NettyStreamManager中的文件与目录缓存

```
private val files = new ConcurrentHashMap[String, File]()
private val jars = new ConcurrentHashMap[String, File]()
private val dirs = new ConcurrentHashMap[String, File]()
```

根据这三个缓存结构，我们猜想 NettyStreamManager 应该可以提供对普通文件、Jar 文件及目录的下载支持。NettyStreamManager 中一共提供了两类方法，一类用于添加缓存，一类用于支持文件流的读取。添加缓存的三个方法（如代码清单 5-19 所示）。

代码清单5-19 添加缓存的方法

```
override def addFile(file: File): String = {
  val existingPath = files.putIfAbsent(file.getName, file)
  require(existingPath == null || existingPath == file,
    s"File ${file.getName} was already registered with a different path " +
      s"(old path = $existingPath, new path = $file")
  s"${rpcEnv.address.toSparkURL}/files/${Utils.encodeFileNameToURIRawPath(file.
      getName())}"
}

override def addJar(file: File): String = {
  val existingPath = jars.putIfAbsent(file.getName, file)
  require(existingPath == null || existingPath == file,
    s"File ${file.getName} was already registered with a different path " +
      s"(old path = $existingPath, new path = $file")
  s"${rpcEnv.address.toSparkURL}/jars/${Utils.encodeFileNameToURIRawPath(file.
      getName())}"
}

override def addDirectory(baseUri: String, path: File): String = {
  val fixedBaseUri = validateDirectoryUri(baseUri)
  require(dirs.putIfAbsent(fixedBaseUri.stripPrefix("/"), path) == null,
    s"URI '$fixedBaseUri' already registered.")
  s"${rpcEnv.address.toSparkURL}$fixedBaseUri"
}
```



代码清单 4-22 中提到的 fileServer 实际就是 NettyStreamManager。由于添加缓存的三个方法都比较简单，所以不再过多介绍。

NettyStreamManager 提供的用于打开文件流的方法如代码清单 5-20 所示。

代码清单5-20 打开文件流

```
override def openStream(streamId: String): ManagedBuffer = {
  val Array(ftype, fname) = streamId.stripPrefix("/").split("/", 2)
  val file = ftype match {
    case "files" => files.get(fname)
    case "jars" => jars.get(fname)
    case other =>
      val dir = dirs.get(ftype)
      require(dir != null, s"Invalid stream URI: $ftype not found.")
      new File(dir, fname)
  }
```

仅供非商业用途或交流学习使用


```
    if (file != null && file.isFile()) {
      new FileSegmentManagedBuffer(rpcEnv.transportConf, file, 0, file.length())
    } else {
      null
    }
  }
```

openStream 方法的逻辑很简单，从缓存中获取文件后，将 TransportConf 及 File 等信息封装为 FileSegmentManagedBuffer 并返回。我们之前在代码清单 3-24、代码清单 3-25 及代码清单 3-26 中介绍过 FileSegmentManagedBuffer 的实现，我们知道 FileSegmentManagedBuffer 封装了对文件的流处理细节。由于 NettyStreamManager 只实现了 Stream Manager 的 openStream 方法，根据 3.2.6 节对 TransportRequestHandler 的 handle 方法和 process StreamRequest 方法的介绍，我们知道 NettyStreamManager 将只提供对 StreamRequest 类型消息的处理。各个 Executor 节点就可以使用 Driver 节点的 RpcEnv 提供的 NettyStreamManager，从 Driver 将 Jar 包或文件下载到 Executor 节点上供任务执行。

2. NettyRpcHandler 详解

上文谈到 NettyRpcEnv 中用于构造 TransportContext 的 RpcHandler 实际是其实现类 NettyRpcHandler。结合代码清单 3-27 中对 RpcHandler 的定义，我们选择两个重载的 receive 方法来看看 NettyRpcHandler 是如何实现 RpcHandler 的。

- 对客户端进行响应的 receive 方法（见代码清单 5-21）的分析。



代码清单5-21　NettyRpcHandler的receive方法

```
override def receive(
    client: TransportClient,
    message: ByteBuffer,
    callback: RpcResponseCallback): Unit = {
  val messageToDispatch = internalReceive(client, message)
  dispatcher.postRemoteMessage(messageToDispatch, callback)
}
```

可以看到，代码清单 5-21 中 receive 方法的处理步骤如下。

1）调用 internalReceive 方法将 ByteBuffer 类型的 message 转换为 RequestMessage。

2）调用 Dispatcher 的 postRemoteMessage 方法（见代码清单 5-15）将消息转换为 RpcMessage 后放入 Inbox 的消息列表。根据代码清单 5-7，MessageLoop 将调用 RpcEndPoint 实现类的 receiveAndReply 方法，即 RpcEndPoint 处理完消息后会向客户端进行回复。

internalReceive 方法可以将 ByteBuffer 类型的 message 转换为 RequestMessage，其实现如代码清单 5-22 所示。

代码清单5-22　internalReceive的实现

```
private def internalReceive(client: TransportClient, message: ByteBuffer):
    RequestMessage = {
  val addr = client.getChannel().remoteAddress().asInstanceOf[InetSocketAddress]
```

仅供非商业用途或交流学习使用


```
    assert(addr != null)
    val clientAddr = RpcAddress(addr.getHostString, addr.getPort)
    val requestMessage = nettyEnv.deserialize[RequestMessage](client, message)
    if (requestMessage.senderAddress == null) {
      RequestMessage(clientAddr, requestMessage.receiver, requestMessage.content)
    } else {
      val remoteEnvAddress = requestMessage.senderAddress
      if (remoteAddresses.putIfAbsent(clientAddr, remoteEnvAddress) == null) {
        dispatcher.postToAll(RemoteProcessConnected(remoteEnvAddress))
      }
      requestMessage
    }
  }
```

根据代码清单 5-22，internalReceive 的执行步骤如下。

1）从 TransportClient 中获取远端地址 RpcAddress。

2）调用 NettyRpcEnv 的 deserialize 方法对客户端发送的序列化后的消息（即 ByteBuffer 类型的消息）进行反序列化，根据 deserialize 的实现（见代码清单 5-23），反序列化实际使用了 javaSerializerInstance。javaSerializerInstance 是通过 NettyRpcEnv 的构造参数传入的对象，类型为 JavaSerializerInstance（请参阅代码清单 5-1）。

代码清单5-23　反序列化ByteBuffer类型的消息

```
private[netty] def deserialize[T: ClassTag](client: TransportClient, bytes:
ByteBuffer): T = {
  NettyRpcEnv.currentClient.withValue(client) {
    deserialize { () =>
      javaSerializerInstance.deserialize[T](bytes)
    }
  }
}
```



3）如果反序列化得到的请求消息 requestMessage 中没有发送者的地址信息，则使用从 TransportClient 中获取的远端地址 RpcAddress、requestMessage 的接收者（即 RpcEndpoint）、requestMessage 的内容，以构造新的 RequestMessage。

4）如果反序列化得到的请求消息 requestMessage 中含有发送者的地址信息，则将从 TransportClient 中获取的远端地址 RpcAddress 与 requestMessage 中的发送者地址信息之间的映射关系放入缓存 remoteAddresses。还将调用 Dispatcher 的 postToAll 方法（见代码清单 5-15）向 endpoints 缓存的所有 EndpointData 的 Inbox 中放入 RemoteProcessConnected 消息[⊖]。最后将返回 requestMessage。

- 对客户端进行响应的 receive 重载方法（其实现见代码清单 5-24）的分析。

⊖ 根据代码清单 5-7 中 Inbox 的 process 方法对 RemoteProcessConnected 消息的匹配，将执行所有 RpcEndpoint 的 onConnected 方法。特质 RpcEndpoint 中的 onConnected 是个空方法，所以 RpcEndpoint 的具体实现类可以针对远端客户端连接到服务端设计一些定制化的功能。

仅供非商业用途或交流学习使用

代码清单5-24 NettyRpcHandler的receive重载方法

```
override def receive(
    client: TransportClient,
    message: ByteBuffer): Unit = {
  val messageToDispatch = internalReceive(client, message)
  dispatcher.postOneWayMessage(messageToDispatch)
}
```

根据 3.2.6 节对 RpcHandler 实现的分析，我们知道此方法不会对客户端进行回复。此方法也调用了 internalReceive 方法，但是最后向 EndpointData 的 Inbox 投递消息使用了 postOneWayMessage 方法（见代码清单 5-15），将消息转换为 OneWayMessage 后放入 Inbox 的消息列表。根据代码清单 5-7，将调用 RpcEndPoint 实现类的 receive 方法，也就是说 RpcEndPoint 处理完消息后不会向客户端进行回复。

由于 NettyRpcHandler 实现了 RpcHandler 的两个 receive 方法，所以根据 3.2.6 节 TransportRequestHandler 的 handle 方法、processRpcRequest 方法及 processOneWayMessage 方法的介绍，我们知道 NettyRpcHandler 将提供对 OneWayMessage 和 RpcRequest 两种类型消息的处理。

NettyRpcHandler 除实现了 RpcHandler 的两个 receive 方法，还实现了 exceptionCaught、channelActive 与 channelInactive 等。exceptionCaught 方法将会向 Inbox 中投递 RemoteProcessConnectionError 消息。channelActive 将会向 Inbox 中投递 RemoteProcessConnected。channelInactive 将会向 Inbox 中投递 RemoteProcessDisconnected 消息。这几个方法的处理都与 receive 方法类似，感兴趣的读者可以自行分析，此处不再赘述。



5.3.6 创建传输客户端工厂 TransportClientFactory

创建传输客户端工厂 TransportClientFactory 是 NettyRpcEnv 向远端服务发起请求的基础，Spark 与远端 RpcEnv 进行通信都依赖于 TransportClientFactory 生产的 TransportClient。NettyRpcEnv 中共创建了两个 TransportClientFactory，代码如下。

```
private def createClientBootstraps(): java.util.List[TransportClientBootstrap] =
{
  if (securityManager.isAuthenticationEnabled()) {
      java.util.Arrays.asList(new SaslClientBootstrap(transportConf, "",
securityManager,
      securityManager.isSaslEncryptionEnabled()))
  } else {
    java.util.Collections.emptyList[TransportClientBootstrap]
  }
}
private val clientFactory = transportContext.createClientFactory(createClientBoo
tstraps())
@volatile private var fileDownloadFactory: TransportClientFactory = _
```

这里的 clientFactory 用于常规的发送请求和接收响应。fileDownloadFactory 则用于文件下载。由于有些 RpcEnv 本身并不需要从远端下载文件，所以这里只声明了变量 fileDownloadFactory，并未进一步对其初始化。需要下载文件的 RpcEnv 会调用 downloadClient 方法创

仅供非商业用途或交流学习使用

建 TransportClientFactory，并用此 TransportClientFactory 创建下载所需的传输客户端 TransportClient。downloadClient 的创建见代码清单 5-25。

代码清单5-25 获取下载的TransportClient

```
private def downloadClient(host: String, port: Int): TransportClient = {
  if (fileDownloadFactory == null) synchronized {
    if (fileDownloadFactory == null) {
      val module = "files"
      val prefix = "spark.rpc.io."
      val clone = conf.clone()

      // Copy any RPC configuration that is not overridden in the spark.files
         namespace.
      conf.getAll.foreach { case (key, value) =>
        if (key.startsWith(prefix)) {
          val opt = key.substring(prefix.length())
          clone.setIfMissing(s"spark.$module.io.$opt", value)
        }
      }

      val ioThreads = clone.getInt("spark.files.io.threads", 1)
      val downloadConf = SparkTransportConf.fromSparkConf(clone, module,
         ioThreads)
      val downloadContext = new TransportContext(downloadConf, new
         NoOpRpcHandler(), true)
      fileDownloadFactory = downloadContext.createClientFactory(createClientBoot
         straps())
    }
  }
  fileDownloadFactory.createClient(host, port)
}
```

根据代码清单 5-25，我们知道 fileDownloadFactory 与 clientFactory 使用的 SparkTransportConf 内部代理的 SparkConf 都是从 NettyRpcEnv 的 SparkConf 克隆来的，不同之处在于 clientFactory 所属的模块（module 变量）为 rpc，fileDownloadFactory 所属的模块为 files。clientFactory 中的读写线程数由 spark.rpc.io.numConnectionsPerPeer 属性控制，而 fileDownloadFactory 中的读写线程数由 spark.files.io.threads 属性控制。有关 SparkTransportConf.fromSparkConf 方法在 3.2.1 节已经介绍过，TransportContext 的 createClientFactory 方法的内容也在 3.2.2 节详细说明，故此处不再赘述。

5.3.7 创建 TransportServer

作为一个 RPC 环境，NettyRpcEnv 不应该只具有向远端服务发起请求并接收响应的能力，也应当对外提供接收请求、处理请求、回复客户端的服务。NettyRpcEnv 中创建 TransportServer 的代码如下[⊖]。

⊖ 这里顺带列出了 NettyRpcEnv 的状态属性。

仅供非商业用途或交流学习使用


```
@volatile private var server: TransportServer = _
private val stopped = new AtomicBoolean(false)
```

可以看到，TransportServer 在此时并未实例化，那么它是何时实例化的？在代码清单 5-1 中我们介绍过启动 RpcEnv 的偏函数 startNettyRpcEnv，startNettyRpcEnv 将负责回调 NettyRpcEnv 的 startServer 方法，startServer 的实现如代码清单 5-26 所示。

代码清单5-26　启动TransportServer

```
def startServer(bindAddress: String, port: Int): Unit = {
  val bootstraps: java.util.List[TransportServerBootstrap] =
    if (securityManager.isAuthenticationEnabled()) {
      java.util.Arrays.asList(new SaslServerBootstrap(transportConf, securityManager))
    } else {
      java.util.Collections.emptyList()
    }
  server = transportContext.createServer(bindAddress, port, bootstraps)
  dispatcher.registerRpcEndpoint(
    RpcEndpointVerifier.NAME, new RpcEndpointVerifier(this, dispatcher))
}
```

根据代码清单 5-26，启动 NettyRpcEnv 的步骤如下。

1）创建 TransportServer。这里使用了 TransportContext 的 createServer 方法，此方法已经在 3.2.3 节详细介绍，不再赘述。

2）向 Dispatcher 注册 RpcEndpointVerifier。RpcEndpointVerifier 用于校验指定名称的 RpcEndpoint 是否存在。RpcEndpointVerifier 在 Dispatcher 中的注册名为 endpoint-verifier，其实现如代码清单 5-27 所示。registerRpcEndpoint 在代码清单 5-9 有详细分析，读者可以回顾。

代码清单5-27　RpcEndpointVerifier的实现

```
private[netty] class RpcEndpointVerifier(override val rpcEnv: RpcEnv, dispatcher:
    Dispatcher)
  extends RpcEndpoint {

  override def receiveAndReply(context: RpcCallContext): PartialFunction[Any,
      Unit] = {
  case RpcEndpointVerifier.CheckExistence(name) => context.reply(dispatcher.
      verify(name))
  }
}

private[netty] object RpcEndpointVerifier {
  val NAME = "endpoint-verifier"
  case class CheckExistence(name: String)
}
```

可以看到 RpcEndpointVerifier 实现了 RpcEndpoint 的 receiveAndReply 方法。因此 MessageLoop 线程在处理 RpcEndpointVerifier 所关联的 Inbox 中的消息时，会匹配 RpcMessage 调用 RpcEndpointVerifier 的 receiveAndReply 方法。RpcEndpointVerifier 实现的

仅供非商业用途或交流学习使用

receiveAndReply 方法的处理步骤如下。

1）接收 CheckExistence 类型的消息，匹配出 name 参数，此参数代表要查询的 RpcEndpoint 的具体名称。

2）调用 Dispatcher 的 verify 方法。verify 用于校验 Dispatcher 的 endpoints 缓存中是否存在名为 name 的 RpcEndpoint，其实现如代码清单 5-28 所示。

3）调用 RpcCallContext 的 reply 方法回复客户端。RpcCallContext 的定义如代码清单 5-16 所示。

代码清单5-28 Dispatcher的verify方法

```
def verify(name: String): Boolean = {
  endpoints.containsKey(name)
}
```

根据对 RpcEndpointVerifier 的实现分析，我们知道它对外提供了查询当前 RpcEndpointVerifier 所在 RpcEnv 的 Dispatcher 中是否存在请求中指定名称所对应的 RpcEndpoint。

TransportServer 初始化并且启动后，就可以利用 NettyRpcHandler 和 NettyStreamManager 对外提供服务了。

5.3.8 客户端请求发送



之前已经介绍了 NettyRpcHandler 和 NettyStreamManager 提供的服务端实现，现在我们来看看如何向远端 RpcEndpoint 发送消息。

当 TransportClient 发出请求之后，会等待获取服务端的回复，这就涉及超时问题。另外由于 TransportClientFactory.createClient 方法是阻塞式调用，所以需要一个异步的处理。NettyRpcEnv 中实现这些需求的代码如下。

```
val timeoutScheduler = ThreadUtils.newDaemonSingleThreadScheduledExecutor("netty-
   rpc-env-timeout")
private[netty] val clientConnectionExecutor = ThreadUtils.newDaemonCached-
    ThreadPool(
  "netty-rpc-connection",
conf.getInt("spark.rpc.connect.threads", 64))
private val outboxes = new ConcurrentHashMap[RpcAddress, Outbox]()
```

上面代码中创建了与发送请求相关的三个组件，分别如下。

- timeoutScheduler：用于处理请求超时的调度器。timeoutScheduler 的类型实际是 ScheduledExecutorService，比起使用 Timer 组件，ScheduledExecutorService 将比 Timer 更加稳定，比如线程挂掉后，ScheduledExecutorService 会重启一个新的线程定时检查请求是否超时。
- clientConnectionExecutor：一个用于异步处理 TransportClientFactory.createClient 方法调用的线程池。这个线程池的大小默认为 64，可以使用 spark.rpc.connect.threads 属性进行配置。

仅供非商业用途或交流学习使用

- outboxes：RpcAddress 与 Outbox 的映射关系的缓存。每次向远端发送请求时，此请求消息首先放入此远端地址对应的 Outbox，然后使用线程异步发送。

1. Outbox 与 OutboxMessage

上文提到的 outboxes 缓存了远端 RPC 地址与 Outbox 的关系，那么 Outbox 是什么？为了说明 Outbox，不妨来看看 Outbox 中的成员属性。

- nettyEnv：当前 Outbox 所在节点上的 NettyRpcEnv。
- address：Outbox 所对应的远端 NettyRpcEnv 的地址。
- messages：向其他远端 NettyRpcEnv 上的所有 RpcEndpoint 发送的消息列表。
- client：当前 Outbox 内的 TransportClient。消息的发送都依赖于此传输客户端。
- connectFuture：指向当前 Outbox 内连接任务的 java.util.concurrent.Future 引用。如果当前没有连接任务，则 connectFuture 为 null。
- stopped：当前 Outbox 是否停止的状态。
- draining：表示当前 Outbox 内正有线程在处理 messages 列表中消息的状态。

消息列表 messages 中的消息类型为 OutboxMessage，所有将要向远端发送的消息都会被封装成 OutboxMessage 类型。OutboxMessage 作为一个特质，定义了所有向外发送消息的规范，其定义如代码清单 5-29 所示。



代码清单5-29　OutboxMessage的定义

```
private[netty] sealed trait OutboxMessage {
  def sendWith(client: TransportClient): Unit
  def onFailure(e: Throwable): Unit
}
```

根据 OutboxMessage 的名称，我们很容易与 Dispatcher 中 Inbox 里的 InboxMessage 类型的消息关联起来。OutboxMessage 在客户端使用，是对外发送消息的封装。InboxMessage 在服务端使用，是对所接收消息的封装。OutboxMessage 的继承体系如图 5-9 所示。

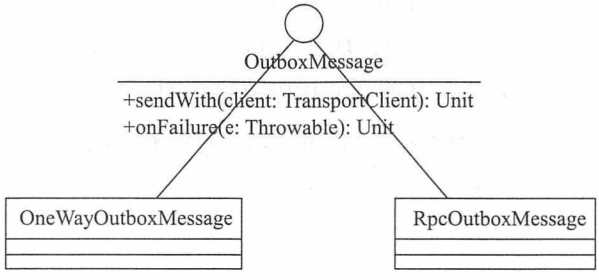


图 5-9　OutboxMessage 的继承体系

我们以 RpcOutboxMessage 为例，来看看 OutboxMessage 该如何实现，RpcOutbox-Message 的实现如代码清单 5-30 所示。

仅供非商业用途或交流学习使用

代码清单5-30 RpcOutboxMessage的实现

```
private[netty] case class RpcOutboxMessage(
  content: ByteBuffer,
  _onFailure: (Throwable) => Unit,
  _onSuccess: (TransportClient, ByteBuffer) => Unit)
extends OutboxMessage with RpcResponseCallback with Logging {

  private var client: TransportClient = _
  private var requestId: Long = _

  override def sendWith(client: TransportClient): Unit = {
    this.client = client
    this.requestId = client.sendRpc(content, this)
  }

  def onTimeout(): Unit = {
    if (client != null) {
      client.removeRpcRequest(requestId)
    } else {
      logError("Ask timeout before connecting successfully")
    }
  }

  override def onFailure(e: Throwable): Unit = {
    _onFailure(e)
  }

  override def onSuccess(response: ByteBuffer): Unit = {
    _onSuccess(client, response)
  }

}
```

根据代码清单 5-30，我们知道 RpcOutboxMessage 重写的 sendWith 方法正是利用了 TransportClient 的 sendRpc 方法（见 3.2.8 节）。TransportClient 的 sendRpc 方法的第二个参数是 RpcResponseCallback 类型，RpcOutboxMessage 本身也实现了 RpcResponseCallback，所以调用的时候传递了 RpcOutboxMessage 的 this 引用。

有了对 Outbox 中各个属性的介绍，现在一起来认识下 Outbox 的各个方法。Outbox 的发送消息的方法是最常用的方法，其实现如代码清单 5-31 所示。

代码清单5-31 Outbox的发送消息的方法

```
def send(message: OutboxMessage): Unit = {
  val dropped = synchronized {
    if (stopped) {
      true
    } else {
      messages.add(message)
      false
    }
  }
```

仅供非商业用途或交流学习使用


```
    if (dropped) {
      message.onFailure(new SparkException("Message is dropped because Outbox is
          stopped"))
    } else {
      drainOutbox()
    }
  }
```

根据代码清单 5-31，Outbox 的 send 方法的执行步骤如下。

1）判断当前 Outbox 的状态是否已经停止。

2）如果 Outbox 已经停止，则向发送者发送 SparkException 异常。如果 Outbox 还未停止，则将 OutboxMessage 添加到 messages 列表中，并且调用 drainOutbox 方法处理 messages 中的消息。drainOutbox 是一个私有方法，其实现如代码清单 5-32 所示。

代码清单5-32　drainOutbox的实现

```
  private def drainOutbox(): Unit = {
    var message: OutboxMessage = null
    synchronized {
      if (stopped) {
        return
      }
      if (connectFuture != null) {
        return
      }
      if (client == null) {
        launchConnectTask()
        return
      }
      if (draining) {
        return
      }
      message = messages.poll()
      if (message == null) {
        return
      }
      draining = true
    }
    while (true) {
      try {
        val _client = synchronized { client }
        if (_client != null) {
          message.sendWith(_client)
        } else {
          assert(stopped == true)
        }
      } catch {
        case NonFatal(e) =>
          handleNetworkFailure(e)
          return
      }
      synchronized {
        if (stopped) {
```



仅供非商业用途或交流学习使用


```
            return
          }
          message = messages.poll()
          if (message == null) {
            draining = false
            return
          }
        }
      }
    }
```

根据代码清单 5-32，drainOutbox 的执行步骤如下。

1）如果当前 Outbox 已经停止或者正在连接远端服务，则返回。

2）如果 Outbox 中的 TransportClient 为 null，这说明还未连接远端服务。此时需要调用 launchConnectTask 方法运行连接远端服务的任务，然后返回。

3）如果正有线程在处理（即发送）messages 列表中的消息，则返回。

4）如果 messages 列表中没有消息要处理，则返回。否则取出其中的一条消息，并将 draining 状态置为 true。

5）循环处理 messages 列表中的消息，即不断从 messages 列表中取出消息并调用 OutboxMessage 的 sendWith 方法发送消息。

在 drainOutbox 方法中调用 launchConnectTask 方法，运行连接远端服务的任务，其实现如代码清单 5-33 所示。



代码清单5-33　运行连接远端服务的任务

```
private def launchConnectTask(): Unit = {
  connectFuture = nettyEnv.clientConnectionExecutor.submit(new Callable[Unit] {

    override def call(): Unit = {
      try {
        val _client = nettyEnv.createClient(address)
        outbox.synchronized {
          client = _client
          if (stopped) {
            closeClient()
          }
        }
      } catch {
        case ie: InterruptedException =>
          // exit
          return
        case NonFatal(e) =>
          outbox.synchronized { connectFuture = null }
          handleNetworkFailure(e)
          return
      }
      outbox.synchronized { connectFuture = null }
      // It's possible that no thread is draining now. If we don't drain here, we
          cannot send the
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
        // messages until the next message arrives.
        drainOutbox()
      }
    })
  }
```

根据代码清单 5-33，launchConnectTask 方法的执行步骤如下。

1）构造 Callable 的匿名内部类，此匿名类将调用 NettyRpcEnv 的 createClient 方法创建 TransportClient，然后调用 drainOutbox 方法处理 Outbox 中的消息。

2）使用 NettyRpcEnv 中的 clientConnectionExecutor 提交 Callable 的匿名内部类。

NettyRpcEnv 的 createClient 方法的实现如代码清单 5-34 所示。

代码清单5-34　创建TransportClient

```
private[netty] def createClient(address: RpcAddress): TransportClient = {
  clientFactory.createClient(address.host, address.port)
}
```

从代码清单 5-34 中看到，NettyRpcEnv 的 createClient 方法实际使用了 TransportClientFactory 的 createClient 方法（见 3.2.2 节）。

2. NettyRpcEndpointRef 详解

在 NettyRpcEnv 中，要向远端 RpcEndpoint 发送请求，首先要持有 RpcEndpoint 的引用对象 NettyRpcEndpointRef（类似于 Akka 中 Actor 的 ActorRef）。NettyRpcEndpointRef 是 RpcEndpointRef 的唯一子类。NettyRpcEndpointRef 重写了 RpcEndpointRef 的部分方法，其实现如代码清单 5-35 所示。

代码清单5-35　NettyRpcEndpointRef的实现

```
private[netty] class NettyRpcEndpointRef(
  @transient private val conf: SparkConf,
  endpointAddress: RpcEndpointAddress,
  @transient @volatile private var nettyEnv: NettyRpcEnv)
  extends RpcEndpointRef(conf) with Serializable with Logging {

  @transient @volatile var client: TransportClient = _

  private val _address = if (endpointAddress.rpcAddress != null) endpointAddress
      else null
  private val _name = endpointAddress.name

  override def address: RpcAddress = if (_address != null) _address.rpcAddress
      else null

  private def readObject(in: ObjectInputStream): Unit = {
    in.defaultReadObject()
    nettyEnv = NettyRpcEnv.currentEnv.value
    client = NettyRpcEnv.currentClient.value
  }
```

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用


```
private def writeObject(out: ObjectOutputStream): Unit = {
  out.defaultWriteObject()
}

override def name: String = _name

override def ask[T: ClassTag](message: Any, timeout: RpcTimeout): Future[T] = {
  nettyEnv.ask(RequestMessage(nettyEnv.address, this, message), timeout)
}

override def send(message: Any): Unit = {
  require(message != null, "Message is null")
  nettyEnv.send(RequestMessage(nettyEnv.address, this, message))
}

override def toString: String = s"NettyRpcEndpointRef(${_address})"

def toURI: URI = new URI(_address.toString)

final override def equals(that: Any): Boolean = that match {
  case other: NettyRpcEndpointRef => _address == other._address
  case _ => false
}

final override def hashCode(): Int = if (_address == null) 0 else _address.
    hashCode()
}
```



根据代码清单 5-35，NettyRpcEndpointRef 包含了以下属性。

- client：类型为 TransportClient（TransportClient 的具体内容请参阅 3.2.8 节）。NettyRpcEndpointRef 将利用此 TransportClient 向远端的 RpcEndpoint 发送请求。
- _address：远端 RpcEndpoint 的地址 RpcEndpointAddress。
- _name：远端 RpcEndpoint 的名称。

下面来介绍 NettyRpcEndpointRef 中重写的方法。

- address：返回 _address 属性的值，或返回 null。
- name：返回 _name 属性的值。
- ask[T: ClassTag](message: Any, timeout: RpcTimeout)：首先将 message 封装为 RequestMessage，然后调用 NettyRpcEnv 的 ask 方法。
- send(message: Any)：首先将 message 封装为 RequestMessage，然后调用 NettyRpcEnv 的 send 方法。

NettyRpcEndpointRef 的 ask 方法和 send 方法分别调用了 NettyRpcEnv 的 ask 方法和 send 方法，下面将对它们进行详细介绍。

（1）询问

NettyRpcEnv 重写了抽象类 RpcEnv 的 ask 方法，其实现如代码清单 5-36 所示。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

代码清单5-36 询问方法ask的实现

```
private[netty] def ask[T: ClassTag](message: RequestMessage, timeout: RpcTimeout):
Future[T] = {
  val promise = Promise[Any]()
  val remoteAddr = message.receiver.address

  def onFailure(e: Throwable): Unit = {
    if (!promise.tryFailure(e)) {
      logWarning(s"Ignored failure: $e")
    }
  }

  def onSuccess(reply: Any): Unit = reply match {
    case RpcFailure(e) => onFailure(e)
    case rpcReply =>
      if (!promise.trySuccess(rpcReply)) {
        logWarning(s"Ignored message: $reply")
      }
  }

  try {
    if (remoteAddr == address) {
      val p = Promise[Any]()
      p.future.onComplete {
        case Success(response) => onSuccess(response)
        case Failure(e) => onFailure(e)
      }(ThreadUtils.sameThread)
      dispatcher.postLocalMessage(message, p)
    } else {
      val rpcMessage = RpcOutboxMessage(serialize(message),
        onFailure,
        (client, response) => onSuccess(deserialize[Any](client, response)))
      postToOutbox(message.receiver, rpcMessage)
      promise.future.onFailure {
        case _: TimeoutException => rpcMessage.onTimeout()
        case _ =>
      }(ThreadUtils.sameThread)
    }

    val timeoutCancelable = timeoutScheduler.schedule(new Runnable {
      override def run(): Unit = {
        onFailure(new TimeoutException(s"Cannot receive any reply in ${timeout.
            duration}"))
      }
    }, timeout.duration.toNanos, TimeUnit.NANOSECONDS)
    promise.future.onComplete { v =>
      timeoutCancelable.cancel(true)
    }(ThreadUtils.sameThread)
  } catch {
    case NonFatal(e) =>
      onFailure(e)
  }
  promise.future.mapTo[T].recover(timeout.addMessageIfTimeout)(ThreadUtils.
      sameThread)
}
```



书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

根据代码清单 5-36，NettyRpcEnv 的 ask 方法的执行步骤如下。

1）如果请求消息的接收者的地址与当前 NettyRpcEnv 的地址相同（则说明处理请求的 RpcEndpoint 位于本地的 NettyRpcEnv 中），那么新建 Promise 对象，并且给 Promise 的 future（类型为 Future）设置完成时的回调函数（成功时调用 onSuccess 方法，失败时调用 onFailure 方法）。发送消息最终通过调用本地 Dispatcher 的 postLocalMessage 方法（见代码清单 5-15）实现。

2）如果请求消息的接收者的地址与当前 NettyRpcEnv 的地址不同（则说明处理请求的 RpcEndpoint 位于其他节点的 NettyRpcEnv 中），那么将 message 序列化，并与 onFailure、onSuccess 方法一道封装为 RpcOutboxMessage 类型的消息。最后调用 postToOutbox 方法将消息投递出去。

3）使用 timeoutScheduler 设置一个定时器，用于超时处理。此定时器在等待指定的超时时间后将抛出 TimeoutException 异常。请求如果在超时时间内处理完毕，则会调用 timeoutScheduler 的 cancel 方法取消此超时定时器。

4）返回请求处理的结果。

（2）发送消息

NettyRpcEnv 重写了抽象类 RpcEnv 的 send 方法，其实现如代码清单 5-37 所示。



代码清单5-37　send的实现

```
private[netty] def send(message: RequestMessage): Unit = {
  val remoteAddr = message.receiver.address
  if (remoteAddr == address) {
    // Message to a local RPC endpoint.
    try {
      dispatcher.postOneWayMessage(message)
    } catch {
      case e: RpcEnvStoppedException => logWarning(e.getMessage)
    }
  } else {
    // Message to a remote RPC endpoint.
    postToOutbox(message.receiver, OneWayOutboxMessage(serialize(message)))
  }
}
```

根据代码清单 5-37，NettyRpcEnv 的 send 方法的执行步骤如下。

1）如果请求消息的接收者的地址与当前 NettyRpcEnv 的地址相同。[⊖]那么新建 Promise 对象，并且给 Promise 的 future（类型为 Future）设置完成时的回调函数（成功时调用 onSuccess 方法，失败时调用 onFailure 方法）。发送消息最终通过调用本地 Dispatcher 的 postOneWayMessage 方法（见代码清单 5-15）实现。

2）如果请求消息的接收者的地址与当前 NettyRpcEnv 的地址不同[⊜]，那么将 message

⊖ 说明处理请求的 RpcEndpoint 位于本地的 NettyRpcEnv 中。

⊜ 说明处理请求的 RpcEndpoint 位于其他节点的 NettyRpcEnv 中。

书籍是人类进步的阶梯

仅供非商业用途或交流学习使用

序列化，并与 onFailure、onSuccess 方法一道封装为 RpcOutboxMessage 类型的消息。最后调用 postToOutbox 方法将消息投递出去。

NettyRpcEnv 的 ask 和 send 方法都调用了私有方法 postToOutbox，postToOutbox 用于向远端节点上的 RpcEndpoint 发送消息，其实现如代码清单 5-38 所示。

代码清单5-38　postToOutbox的实现

```
private def postToOutbox(receiver: NettyRpcEndpointRef, message: OutboxMessage):
    Unit = {
  if (receiver.client != null) {
    message.sendWith(receiver.client)
  } else {
    require(receiver.address != null,
      "Cannot send message to client endpoint with no listen address.")
    val targetOutbox = {
      val outbox = outboxes.get(receiver.address)
      if (outbox == null) {
        val newOutbox = new Outbox(this, receiver.address)
        val oldOutbox = outboxes.putIfAbsent(receiver.address, newOutbox)
        if (oldOutbox == null) {
          newOutbox
        } else {
          oldOutbox
        }
      } else {
        outbox
      }
    }
    if (stopped.get) {
      // It's possible that we put 'targetOutbox' after stopping. So we need to clean it.
      outboxes.remove(receiver.address)
      targetOutbox.stop()
    } else {
      targetOutbox.send(message)
    }
  }
}
```

根据代码清单 5-38，postToOutbox 方法的执行步骤如下。

1）如果 NettyRpcEndpointRef 中的 TransportClient 不为空，则直接调用 Outbox-Message 的 sendWith 方法，否则进入第 2）步。

2）获取 NettyRpcEndpointRef 的远端 RpcEndpoint 地址所对应的 Outbox。首先从 outboxes 缓存中获取 Outbox。如果 outboxes 中没有相应的 Outbox，则需要新建 Outbox 并放入 outboxes 缓存中。

3）如果当前 NettyRpcEnv 已经处于停止状态，则将第 2）步得到的 Outbox 从 outboxes 中移除，并且调用 Outbox 的 stop 方法停止 Outbox。如果当前 NettyRpcEnv 还未停止，则调用第 2）步得到的 Outbox 的 send 方法发送消息。

Outbox 的 stop 方法用于停止 Outbox，其实现如代码清单 5-39 所示。



仅供非商业用途或交流学习使用

代码清单5-39 Outbox的stop方法

```
def stop(): Unit = {
  synchronized {
    if (stopped) {
      return
    }
    stopped = true
    if (connectFuture != null) {
      connectFuture.cancel(true)
    }
    closeClient()
  }

  var message = messages.poll()
  while (message != null) {
    message.onFailure(new SparkException("Message is dropped because Outbox is
        stopped"))
    message = messages.poll()
  }
}
```

根据代码清单 5-39，可以看到停止 Outbox 的效果包括如下几项。

- 将 Outbox 的停止状态 stopped 置为 true。
- 关闭 Outbox 中的 TransportClient。
- 清空 messages 列表中的消息。

根据本节对客户端发送请求的分析，现在我们可以将此流程用图 5-10 表示。

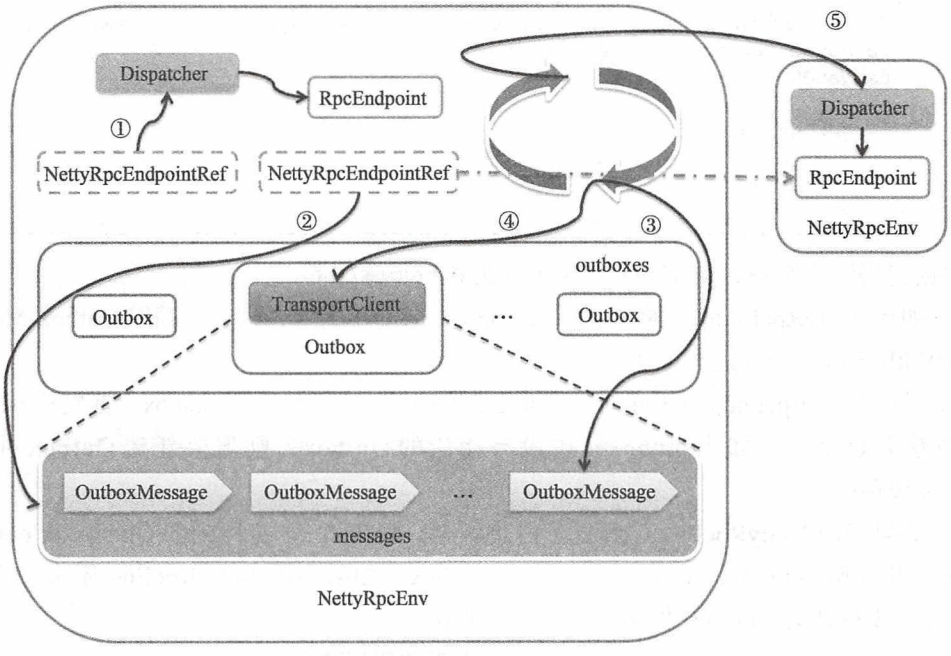


图 5-10 RPC 客户端发送请求示意图

仅供非商业用途或交流学习使用

图 5-10 展示了两个不同节点上的 NettyRpcEnv。右边的 NettyRpcEnv 采用简略的表示方法，只展示了其内部的 Dispatcher 组件，实际上右边的 NettyRpcEnv 与左边的 NettyRpcEnv 在结构和组成上是一样的。除了 Dispatcher 组件，还展示了 NettyRpcEnv 内部的 outboxes 列表、outboxes 列表内的 Outbox 及 Outbox 内部用于缓存 OutboxMessage 的 messages 列表。左边 NettyRpcEnv 中的 NettyRpcEndpointRef 和右边一个提供服务的 RpcEndpoint 之间有箭头的虚线表示 NettyRpcEndpointRef 引用 RpcEndpoint，也就是说 NettyRpcEndpointRef 知道 RpcEndpoint 的地址信息。有了这些简短的陈述，我们可以对图 5-10 中各个序号的含义进行简单的说明。

序号①：表示通过调用 NettyRpcEndpointRef 的 send 和 ask 方法向本地节点的 RpcEndpoint 发送消息。由于是在同一节点，所以直接调用 Dispatcher 的 postLocalMessage 或 postOneWayMessage 方法将消息放入 EndpointData 内部 Inbox 的 messages 列表中。MessageLoop 线程最后处理消息，并将消息发给对应的 RpcEndpoint 处理。

序号②：表示通过调用 NettyRpcEndpointRef 的 send 和 ask 方法向远端节点的 RpcEndpoint 发送消息。这种情况下，消息将首先被封装为 OutboxMessage，然后放入到远端 RpcEndpoint 的地址所对应的 Outbox 的 messages 列表中。

序号③：表示每个 Outbox 的 drainOutbox 方法通过循环，不断从 messages 列表中取得 OutboxMessage。

序号④：表示每个 Outbox 的 drainOutbox 方法使用 Outbox 内部的 TransportClient 向远端的 NettyRpcEnv 发送序号③中取得的 OutboxMessage。

序号⑤：表示序号④发出的请求在与远端 NettyRpcEnv 的 TransportServer 建立了连接后，请求消息首先经过 Netty 管道的处理，然后经由 NettyRpcHandler 处理，最后 NettyRpcHandler 的 receive 方法会调用 Dispatcher 的 postRemoteMessage 或 postOneWayMessage 方法，将消息放入 EndpointData 内部 Inbox 的 messages 列表中。MessageLoop 线程最后处理消息，并将消息发给对应的 RpcEndpoint 处理。

5.3.9　NettyRpcEnv 中的常用方法

在 NettyRpcEnv 中有一些方法经常要用到，所以对它们做一些简单介绍。

1）获取 RpcEndpoint 的引用对象 RpcEndpointRef。NettyRpcEnv 重写了抽象类 RpcEnv 的 endpointRef 方法，其实现如代码清单 5-40 所示。

代码清单5-40　获取RpcEndpoint的引用

```
override def endpointRef(endpoint: RpcEndpoint): RpcEndpointRef = {
  dispatcher.getRpcEndpointRef(endpoint)
}
```

可以看到，endpointRef 方法的实质是调用了 Dispatcher 的 getRpcEndpointRef 方法。getRpcEndpointRef 实际是从 endpointRefs 缓存中获取的 RpcEndpoint 的引用对象 RpcEndpointRef，如代码清单 5-41 所示。

仅供非商业用途或交流学习使用

代码清单5-41 Dispatcher的getRpcEndpointRef方法

```
def getRpcEndpointRef(endpoint: RpcEndpoint): RpcEndpointRef = endpointRefs.
  get(endpoint)
```

2）使用 RpcAddress 和 RpcEndpoint 的名称，得到对应的 RpcEndpointRef。setupEndpointRef 方法实际是 NettyRpcEnv 继承自父类 RpcEnv 的方法，其实现如代码清单 5-42 所示。

代码清单5-42 setupEndpointRef的实现

```
def setupEndpointRef(address: RpcAddress, endpointName: String): RpcEndpointRef = {
  setupEndpointRefByURI(RpcEndpointAddress(address, endpointName).toString)
}
```

可以看到 setupEndpointRef 方法内部调用了 setupEndpointRefByURI，其实现如代码清单 5-43 所示。

代码清单5-43 setupEndpointRefByURI的实现

```
def setupEndpointRefByURI(uri: String): RpcEndpointRef = {
  defaultLookupTimeout.awaitResult(asyncSetupEndpointRefByURI(uri))
}
```

这里的 defaultLookupTimeout 是通过调用 RpcUtils 的 lookupRpcTimeout 方法（详见附录 H）得到的查找超时时长。asyncSetupEndpointRefByURI 的功能是向远端 NettyRpcEnv 询问指定名称的 RpcEndpoint 的 NettyRpcEndpointRef。可以看到这里调用 NettyRpcEndpointRef 的 ask 方法（5.3.8 节对此有详细介绍）向远端 NettyRpcEnv 中的 RpcEndpointVerifier 发送 RpcEndpointVerifier.CheckExistence 类型的消息。RpcEndpointVerifier 的实现已在代码清单 5-27 中详细介绍过。asyncSetupEndpointRefByURI 的实现如代码清单 5-44 所示。



代码清单5-44 异步获取远端NettyRpcEndpointRef

```
def asyncSetupEndpointRefByURI(uri: String): Future[RpcEndpointRef] = {
  val addr = RpcEndpointAddress(uri)
  val endpointRef = new NettyRpcEndpointRef(conf, addr, this)
  val verifier = new NettyRpcEndpointRef(
    conf, RpcEndpointAddress(addr.rpcAddress, RpcEndpointVerifier.NAME), this)
  verifier.ask[Boolean](RpcEndpointVerifier. CheckExistence (endpointRef.name)).
      flatMap { find =>
    if (find) {
      Future.successful(endpointRef)
    } else {
      Future.failed(new RpcEndpointNotFoundException(uri))
    }
  }(ThreadUtils.sameThread)
}
```

- 停止 RpcEndpointRef 所引用的 RpcEndpoint。NettyRpcEnv 重写了抽象类 RpcEnv 的 stop 方法，其实现如代码清单 5-45 所示。

仅供非商业用途或交流学习使用

代码清单5-45　停止RpcEndpoint

```
override def stop(endpointRef: RpcEndpointRef): Unit = {
  require(endpointRef.isInstanceOf[NettyRpcEndpointRef])
  dispatcher.stop(endpointRef)
}
```

可以看到，stop 方法的实质是调用了 Dispatcher 的 stop 方法（见代码清单 5-10）。Dispatcher 的 stop 方法对 RpcEndpoint 去注册，此 RpcEndpoint 相关联的数据（即 EndpointData 关联的 RpcEndpoint、RpcEndpointRef 和 Inbox）都将被移除，RpcEndpoint 也会停止。

4）设置 Endpoint。其实现如代码清单 5-46 所示。

代码清单5-46　设置Endpoint

```
override def setupEndpoint(name: String, endpoint: RpcEndpoint): RpcEndpointRef = {
  dispatcher.registerRpcEndpoint(name, endpoint)
}
```

从代码清单 5-46 可以看到，实际是调用了代码清单 5-9 中 Dispatcher 的 registerRpcEndpoint 方法注册 Endpoint。

5.4　序列化管理器 SerializerManager



Spark 中很多对象在通过网络传输或者写入存储体系时，都需要序列化。SparkEnv 中有两个序列化的组件，分别是 SerializerManager 和 closureSerializer，SparkEnv 中创建它们的代码如下。

```
val serializer = instantiateClassFromConf[Serializer](
    "spark.serializer", "org.apache.spark.serializer.JavaSerializer")
  logDebug(s"Using serializer: ${serializer.getClass}")

  val serializerManager = new SerializerManager(serializer, conf,
    ioEncryptionKey)

val closureSerializer = new JavaSerializer(conf)
```

可以看到这里创建的 serializer 默认为 org.apache.spark.serializer.JavaSerializer，用户可以通过 spark.serializer 属性配置其他的序列化实现，如 org.apache.spark.serializer.KryoSerializer。closureSerializer 的实际类型固定为 org.apache.spark.serializer.JavaSerializer，用户不能够自己指定。JavaSerializer 采用 Java 语言自带的序列化 API 实现，由于 JavaSerializer 的实现非常简单，这里不多介绍，感兴趣的读者可自行阅读其实现。

1. 序列化管理器 SerializerManager 的属性

SerializerManager 给各种 Spark 组件提供序列化、压缩及加密的服务。这里主要对 SerializerManager 中的各个成员属性进行介绍。

仅供非商业用途或交流学习使用

- defaultSerializer：默认的序列化器。此 defaultSerializer 即为上面代码中实例化的 serializer，也就是说 defaultSerializer 的类型是 JavaSerializer。
- conf：即 SparkConf。
- encryptionKey：加密使用的密钥。
- kryoSerializer：Spark 提供的另一种序列化器。kryoSerializer 的实际类型是 KryoSerializer，其采用 Google 提供的 Kryo 序列化库实现。
- stringClassTag：字符串类型标记，即 ClassTag[String]。
- primitiveAndPrimitiveArrayClassTags：原生类型及原生类型数组的类型标记的集合，包括：Boolean、Array[boolean]、Int、Array[int]、Long、Array[long]、Byte、Array[byte]、Null、Array[scala.runtime.Null$]、Char、Array[char]、Double、Array [double]、Float、Array[float]、Short、Array[short] 等。
- compressBroadcast：是否对广播对象进行压缩，可以通过 spark.broadcast.compress 属性配置，默认为 true。
- compressShuffle：是否对 Shuffle 输出数据压缩，可以通过 spark.shuffle.compress 属性配置，默认为 true。
- compressRdds：是否对 RDD 压缩，可以通过 spark.rdd.compress 属性配置，默认为 false。
- compressShuffleSpill：是否对溢出到磁盘的 Shuffle 数据压缩，可以通过 spark.shuffle.spill.compress 属性配置，默认为 true。
- compressionCodec：SerializerManager 使用的压缩编解码器。compressionCodec 的类型是 CompressionCodec。在 Spark 1.x.x 版本中，compressionCodec 是 BlockManager 的成员之一，现在把 compressionCodec 和序列化、加密等功能都集中到 SerializerManager 中，也许是因为实现此功能的工程师觉得加密、压缩都是属于序列化的一部分吧。

2. 创建 CompressionCodec

为了节省磁盘存储空间，有些情况下需要对数据进行压缩。在 SerializerManager 中创建 compressionCodec 的代码如下。

```
private lazy val compressionCodec: CompressionCodec = CompressionCodec.
    createCodec(conf)
```

可以看到 compressionCodec 通过 lazy 关键字修饰为延迟初始化，即等到真正使用时才对其初始化。CompressionCodec 的 createCodec 方法用于创建 CompressionCodec，其实现如代码清单 5-47 所示。

代码清单5-47　CompressionCodec的createCodec方法

```
private val shortCompressionCodecNames = Map(
```

仅供非商业用途或交流学习使用


```
    "lz4" -> classOf[LZ4CompressionCodec].getName,
    "lzf" -> classOf[LZFCompressionCodec].getName,
    "snappy" -> classOf[SnappyCompressionCodec].getName)

  def getCodecName(conf: SparkConf): String = {
    conf.get(configKey, DEFAULT_COMPRESSION_CODEC)
  }

  def createCodec(conf: SparkConf): CompressionCodec = {
    createCodec(conf, getCodecName(conf))
  }

  def createCodec(conf: SparkConf, codecName: String): CompressionCodec = {
    val codecClass = shortCompressionCodecNames.getOrElse(codecName.toLowerCase,
      codecName)
    val codec = try {
      val ctor = Utils.classForName(codecClass).getConstructor(classOf[SparkConf])
      Some(ctor.newInstance(conf).asInstanceOf[CompressionCodec])
    } catch {
      case e: ClassNotFoundException => None
      case e: IllegalArgumentException => None
    }
    codec.getOrElse(throw new IllegalArgumentException(s"Codec [$codecName] is not
        available. " +
      s"Consider setting $configKey=$FALLBACK_COMPRESSION_CODEC"))
  }
```

根据代码清单 5-47，我们看到 createCodec 方法首先调用了 getCodecName 方法获取编解码器的名称。其中的变量 configKey 为 spark.io.compression.codec，即可以使用此属性配置压缩需要的编解码器名称。如果没有指定 spark.io.compression.codec，那么编解码器的默认名称为 lz4（即常量 DEFAULT_COMPRESSION_CODEC 的值）。然后会调用重载的 createCodec 方法，其执行步骤如下。

1）从 shortCompressionCodecNames 缓存中获取编解码器名称对应的编解码器的类名。

2）通过 Java 反射生成编解码器的实例。



在 Spark 1.x.x 版本中，默认的压缩算法为 snappy，snappy 在牺牲少量压缩比例的条件下，极大地提高了压缩速度。

3. 序列化管理器 SerializerManager 的方法

在 SerializerManager 中提供了很多用于序列化、反序列化、压缩、加密的方法，这里对这些功能进行介绍。

- encryptionEnabled：当前 SerializerManager 是否支持加密。SerializerManager 要想支持加密，必须在构造 SerializerManager 的时候就传入 encryptionKey。可以通过 spark.io.encryption.enabled（允许加密）、spark.io.encryption.keySizeBits（密钥长度，有 128、192、256 三种长度）、spark.io.encryption.keygen.algorithm（加密算法，默认为 HmacSHA1）等属性进行具体的配置。

仅供非商业用途或交流学习使用

- canUseKryo(ct: ClassTag[])：对于指定的类型标记 ct，是否能使用 kryoSerializer 进行序列化。当类型标记 ct 属于 primitiveAndPrimitiveArrayClassTags 或者 stringClassTag 时，canUseKryo 方法才返回真。
- getSerializer(ct: ClassTag[], autoPick: Boolean)：获取序列化器。如果 autoPick 为 true（即 BlockId 不为 StreamBlockId 时）并且调用 canUseKryo 的结果为 true 时选择 kryoSerializer，否则选择 defaultSerializer。
- getSerializer(keyClassTag: ClassTag[], valueClassTag: ClassTag[])：获取序列化器。如果对于 keyClassTag 和 valueClassTag，调用 canUseKryo 的结果都为 true 时选择 kryoSerializer，否则选择 defaultSerializer。
- wrapStream(blockId: BlockId, s: InputStream)：对 Block 的输入流进行压缩与加密。
- wrapStream(blockId: BlockId, s: OutputStream)：对 Block 的输出流进行压缩与加密。
- wrapForEncryption(s: InputStream)：对输入流进行加密。
- wrapForEncryption(s: OutputStream)：对输出流进行加密。
- dataSerializeStream[T: ClassTag](blockId: BlockId, outputStream: OutputStream, values: Iterator[T])：对 Block 的输出流序列化。
- dataSerialize[T: ClassTag](blockId: BlockId,values: Iterator[T],allowEncryption: Boolean = true)：序列化成分块字节缓冲区（ChunkedByteBuffer）。
- dataSerializeWithExplicitClassTag(blockId:BlockId,values: Iterator[],classTag: ClassTag[], allowEncryption: Boolean = true)：使用明确的类型标记，序列化成分块字节缓冲区（ChunkedByteBuffer）。
- dataDeserializeStream[T](blockId: BlockId, inputStream: InputStream, maybeEncrypted: Boolean = true) (classTag: ClassTag[T])：将输入流反序列化为值的迭代器 Iterator[T]。

5.5 广播管理器 BroadcastManager

BroadcastManager 用于将配置信息和序列化后的 RDD、Job 及 ShuffleDependency 等信息在本地存储。如果为了容灾，也会复制到其他节点上。创建 BroadcastManager 的代码实现如下。

```
val broadcastManager = new BroadcastManager(isDriver, conf, securityManager)
```

BroadcastManager 除了构造器定义的三个成员属性外，BroadcastManager 内部还有三个成员，分别如下。

- initialized：表示 BroadcastManager 是否初始化完成的状态。
- broadcastFactory：广播工厂实例。
- nextBroadcastId：下一个广播对象的广播 ID，类型为 AtomicLong。

仅供非商业用途或交流学习使用

BroadcastManager 在其初始化的过程中就会调用自身的 initialize 方法，当 initialize 执行完毕，BroadcastManager 就正式生效。BroadcastManager 的 initialize 方法的实现如代码清单 5-48 所示。

代码清单5-48　BroadcastManager的初始化

```
private def initialize() {
  synchronized {
    if (!initialized) {
      broadcastFactory = new TorrentBroadcastFactory
      broadcastFactory.initialize(isDriver, conf, securityManager)
      initialized = true
    }
  }
}
```

根据代码清单 5-48，initialize 方法首先判断 BroadcastManager 是否已经初始化，以保证 BroadcastManager 只被初始化一次。新建 TorrentBroadcastFactory 作为 BroadcastManager 的广播工厂实例。之后调用 TorrentBroadcastFactory 的 initialize 方法对 TorrentBroadcastFactory 进行初始化[⊖]。最后将 BroadcastManager 自身标记为初始化完成状态。



TorrentBroadcastFactory 实现了 BroadcastFactory 特质。在 Spark 1.x.x 版本中，BroadcastManager 的 initialize 方法是使用 Java 反射生成广播工厂实例 broadcastFactory 的，还可以通过配置属性 spark.broadcast.factory 指定 BroadcastFactory 特质的实现类，默认为 org.apache.spark. broadcast.TorrentBroadcastFactory。从 Spark 2.0.0 版本开始，不再提供此 Spark 属性，属性成员 broadcastFactory 也固定为 TorrentBroadcastFactory。



BroadcastManager 中提供了三个方法，如代码清单 5-49 所示。

代码清单5-49　BroadcastManager中的三个方法

```
def stop() {
  broadcastFactory.stop()
}
private val nextBroadcastId = new AtomicLong(0)
def newBroadcast[T: ClassTag](value_ : T, isLocal: Boolean): Broadcast[T] = {
  broadcastFactory.newBroadcast[T](value_, isLocal, nextBroadcastId.getAnd-
      Increment())
}
def unbroadcast(id: Long, removeFromDriver: Boolean, blocking: Boolean) {
  broadcastFactory.unbroadcast(id, removeFromDriver, blocking)
}
```

从代码清单 5-49 可以看到，BroadcastManager 的三个方法都分别代理了 TorrentBroadcast-

⊖ 笔者查阅 TorrentBroadcastFactory 的源码后，发现 TorrentBroadcastFactory 的 Initialize 方法实际是一个空实现，所以这里不再介绍。

仅供非商业用途或交流学习使用

Factory 的对应方法，TorrentBroadcastFactory 中提供的三个方法的实现如代码清单 5-50 所示。

代码清单5-50 TorrentBroadcastFactory提供的方法

```
override def newBroadcast[T: ClassTag](value_ : T, isLocal: Boolean, id: Long):
    Broadcast[T] = {
  new TorrentBroadcast[T](value_, id)
}
override def stop() { }
override def unbroadcast(id: Long, removeFromDriver: Boolean, blocking: Boolean) {
  TorrentBroadcast.unpersist(id, removeFromDriver, blocking)
}
```

代码清单 5-50 中 TorrentBroadcastFactory 提供的三个方法，由于 stop 是空实现，所以我们只关注 newBroadcast 和 unbroadcast 两个方法。

根据代码清单 5-50，我们知道 TorrentBroadcastFactory 的 newBroadcast 方法用于生成 TorrentBroadcast 实例，其作用为广播 TorrentBroadcast 中的 value。表面看只是利用构造器生成了 TorrentBroadcast 实例，但是其效果远不止此。TorrentBroadcast 对象包括以下属性。

- compressionCodec：用于广播对象的压缩编解码器。可以设置 spark.broadcast.compress 属性为 true 启用，默认是启用的。compressionCodec 的类型为 5.4 节详细介绍过的 CompressionCodec，而且最终采用的压缩算法与 SerializerManager 中的 CompressionCodec 是一致的。
- blockSize：每个块的大小。它是个只读属性，可以使用 spark.broadcast.blockSize 属性进行配置，默认为 4MB。
- broadcastId：广播 Id。broadcastId 实际是样例类 BroadcastBlockId，BroadcastBlockId 类的实现如下。

```
case class BroadcastBlockId(broadcastId: Long, field: String = "") extends
    BlockId {
  override def name: String = "broadcast_" + broadcastId + (if (field == "") ""
      else "_" + field)
}
```

其中，broadcastId 是由 BroadcastManager 的原子变量 nextBroadcastId 自增产生的。

- checksumEnabled：是否给广播块生成校验和。可以通过 spark.broadcast.checksum 属性进行配置，默认为 true。
- checksums：用于存储每个广播块的校验和的数组。
- numBlocks：广播变量包含的块的数量。numBlocks 通过调用 writeBlocks 方法获得。由于 numBlocks 是个 val 修饰的不可变属性，因此在构造 TorrentBroadcast 实例的时候就会调用 writeBlocks 方法将广播对象写入存储体系。
- _value：从 Executor 或者 Driver 上读取的广播块的值。_value 是通过调用 readBroadcastBlock 方法获得的广播对象。由于 _value 是个 lazy 及 val 修饰的属性，因



仅供非商业用途或交流学习使用

此在构造 TorrentBroadcast 实例的时候不会调用 readBroadcastBlock 方法，而是等到明确需要使用 _value 的值时。

1. 广播对象的写操作

刚才说到在构造 TorrentBroadcast 实例的时候就会调用 writeBlocks 方法，其实现如代码清单 5-51 所示。

代码清单5-51　writeBlocks的实现

```
private def writeBlocks(value: T): Int = {
  import StorageLevel._
  val blockManager = SparkEnv.get.blockManager
  if (!blockManager.putSingle(broadcastId, value, MEMORY_AND_DISK, tellMaster =
      false)) {
    throw new SparkException(s"Failed to store $broadcastId in BlockManager")
  }
  val blocks =
    TorrentBroadcast.blockifyObject(value, blockSize, SparkEnv.get.serializer,
        compressionCodec)
  if (checksumEnabled) {
    checksums = new Array[Int](blocks.length)
  }
  blocks.zipWithIndex.foreach { case (block, i) =>
    if (checksumEnabled) {
      checksums(i) = calcChecksum(block)
    }
    val pieceId = BroadcastBlockId(id, "piece" + i)
    val bytes = new ChunkedByteBuffer(block.duplicate())
    if (!blockManager.putBytes(pieceId, bytes, MEMORY_AND_DISK_SER, tellMaster =
        true)) {
      throw new SparkException(s"Failed to store $pieceId of $broadcastId in
          local BlockManager")
    }
  }
  blocks.length
}
```

根据代码清单 5-51，writeBlocks 的执行步骤如下。

1）获取当前 SparkEnv 的 BlockManager 组件。

2）调用 BlockManager 的 putSingle 方法将广播对象写入本地的存储体系。当 Spark 以 local 模式运行时，则会将广播对象写入 Driver 本地的存储体系，以便于任务也可以在 Driver 上执行。由于 MEMORY_AND_DISK 对应的 StorageLevel 的 _replication 属性固定为 1，因此此处只会将广播对象写入 Driver 或 Executor 本地的存储体系，有关 BlockManager 的 putSingle 方法及 StorageLevel 的内容将在第 6 章详细介绍。

3）调用 TorrentBroadcast 的 blockifyObject 方法（实现很简单，感兴趣的读者可以自行查阅），将对象转换成一系列的块。每个块的大小由 blockSize 决定，使用当前 SparkEnv 中的 JavaSerializer 组件进行序列化，使用 TorrentBroadcast 自身的 compressionCodec 进行压缩。

4）如果需要给分片广播块生成校验和，则创建和第 3）步转换的块的数量一致的 checksums 数组。

5）对每个块进行如下处理：如果需要给分片广播块生成校验和，则给分片广播块生成校验和。calcChecksum 方法的实现很简单，感兴趣的读者可以自行查阅。给当前分片广播块生成分片的 BroadcastBlockId，分片通过 BroadcastBlockId 的 field 属性区别，例如，piece0、piece1……调用 BlockManager 的 putBytes 方法将分片广播块以序列化方式写入 Driver 本地的存储体系。由于 MEMORY_AND_DISK_SER 对应的 StorageLevel 的 _replication 属性也固定为 1，因此此处只会将分片广播块写入 Driver 或 Executor 本地的存储体系，有关 BlockManager 的 putBytes 方法及 StorageLevel 的内容将在第 6 章详细介绍。

6）返回块的数量。

经过以上分析，最后用图 5-11 来更直观地表示广播对象的写入过程。

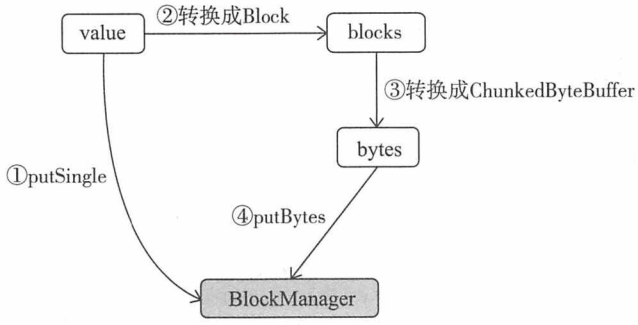


图 5-11　广播对象的写入

2. 广播对象的读操作

前文提到，只有当 TorrentBroadcast 实例的 _value 属性值在需要的时候，才会调用 readBroadcastBlock 方法获取值，readBroadcastBlock 的实现如代码清单 5-52 所示。

代码清单5-52　readBroadcastBlock的实现

```
private def readBroadcastBlock(): T = Utils.tryOrIOException {
  TorrentBroadcast.synchronized {
    setConf(SparkEnv.get.conf)
    val blockManager = SparkEnv.get.blockManager
    blockManager.getLocalValues(broadcastId) match {
      case Some(blockResult) =>
        if (blockResult.data.hasNext) {
          val x = blockResult.data.next().asInstanceOf[T]
          releaseLock(broadcastId)
          x
        } else {
          throw new SparkException(s"Failed to get locally stored broadcast data:
              $broadcastId")
        }
      case None =>
```

```
        logInfo("Started reading broadcast variable " + id)
        val startTimeMs = System.currentTimeMillis()
        val blocks = readBlocks().flatMap(_.getChunks())
        logInfo("Reading broadcast variable " + id + " took" + Utils.
            getUsedTimeMs(startTimeMs))

        val obj = TorrentBroadcast.unBlockifyObject[T](
          blocks, SparkEnv.get.serializer, compressionCodec)
        val storageLevel = StorageLevel.MEMORY_AND_DISK
        if (!blockManager.putSingle(broadcastId, obj, storageLevel, tellMaster =
            false)) {
          throw new SparkException(s"Failed to store $broadcastId in Block-
              Manager")
        }
        obj
    }
  }
}
```

根据代码清单 5-52，readBroadcastBlock 的执行步骤如下。

1）获取当前 SparkEnv 的 BlockManager 组件。

2）调用 BlockManager 的 getLocalValues 方法从本地的存储系统中获取广播对象，即通过 BlockManager 的 putSingle 方法写入存储体系的广播对象，BlockManager 的 getLocalValues 方法将在 6.7.2 节详细介绍。

3）如果从本地的存储体系中可以获取广播对象，则调用 releaseLock 方法（这个锁保证当块被一个运行中的任务使用时，不能被其他任务再次使用，但是当任务运行完成时，则应该释放这个锁），释放当前块的锁并返回此广播对象。

4）如果从本地的存储体系中没有获取到广播对象，那么说明数据是通过 BlockManager 的 putBytes 方法以序列化方式写入存储体系的。此时首先调用 readBlocks 方法从 Driver 或 Executor 的存储体系中获取广播块，然后调用 TorrentBroadcast 的 unBlockifyObject 方法（实现很简单，感兴趣的读者可以自行查阅），将一系列的分片广播块转换回原来的广播对象，最后再次调用 BlockManager 的 putSingle 方法将广播对象写入本地的存储体系，以便于当前 Executor 的其他任务不用再次获取广播对象。

上文谈到调用 readBlocks 方法可以从 Driver、Executor 的存储体系中获取块，其实现如代码清单 5-53 所示。

代码清单5-53　readBlocks的实现

```
private def readBlocks(): Array[ChunkedByteBuffer] = {
  val blocks = new Array[ChunkedByteBuffer](numBlocks)
  val bm = SparkEnv.get.blockManager
  for (pid <- Random.shuffle(Seq.range(0, numBlocks))) {
    val pieceId = BroadcastBlockId(id, "piece" + pid)
    logDebug(s"Reading piece $pieceId of $broadcastId")
    bm.getLocalBytes(pieceId) match {
      case Some(block) =>
```

```
          blocks(pid) = block
          releaseLock(pieceId)
        case None =>
          bm.getRemoteBytes(pieceId) match {
            case Some(b) =>
              if (checksumEnabled) {
                val sum = calcChecksum(b.chunks(0))
                if (sum != checksums(pid)) {
                  throw new SparkException(s"corrupt remote block $pieceId of
                      $broadcastId:" +
                    s" $sum != ${checksums(pid)}")
                }
              }
              // We found the block from remote executors/driver's BlockManager, so
                  put the block
              // in this executor's BlockManager.
              if (!bm.putBytes(pieceId, b, StorageLevel.MEMORY_AND_DISK_SER, tellMaster
                  = true)) {
                throw new SparkException(
                  s"Failed to store $pieceId of $broadcastId in local Block-
                      Manager")
              }
              blocks(pid) = b
            case None =>
              throw new SparkException(s"Failed to get $pieceId of $broadcastId")
          }
      }
    }
    blocks
  }
```

根据代码清单 5-53，readBlocks 方法的执行步骤如下。

1）新建用于存储每个分片广播块的数组 blocks，并获取当前 SparkEnv 的 BlockManager 组件。

2）对各个广播分片进行随机洗牌，避免对广播块的获取出现“热点”，提升性能。对洗牌后的各个广播分片依次执行 3）至 5）步的操作。

3）调用 BlockManager 的 getLocalBytes 方法从本地的存储体系中获取序列化的分片广播块，如果本地可以获取到，则将分片广播块放入 blocks，并且调用 releaseLock 方法释放此分片广播块的锁。

4）如果本地没有，则调用 BlockManager 的 getRemoteBytes 方法从远端的存储体系中获取分片广播块。对于获取的分片广播块再次调用 calcChecksum 方法计算校验和，并将此校验和与调用 writeBlocks 方法时存入 checksums 数组的校验和进行比较。如果校验和不相同，说明块的数据有损坏，此时抛出异常。

5）如果校验和相同，则调用 BlockManager 的 putBytes 方法将分片广播块写入本地存储体系，以便于当前 Executor 的其他任务不用再次获取分片广播块。最后将分片广播块放入 blocks。

6）返回 blocks 中的所有分片广播块。

经过以上的分析，最后用图 5-12 来更直观地表示广播对象的读取过程。

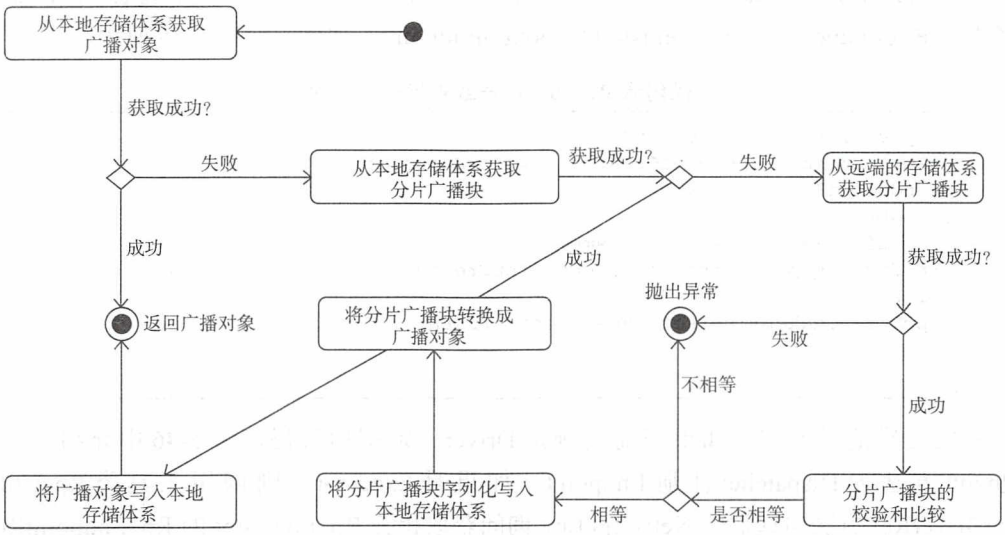


图 5-12　广播对象的读取

3. 广播对象的去持久化

根据代码清单 5-50，我们知道 TorrentBroadcastFactory 的 unbroadcast 方法实际调用了 TorrentBroadcast 的 unpersist 方法对由 id 标记的广播对象去持久化。TorrentBroadcast 的 unpersist 方法的实现如代码清单 5-54 所示。

代码清单5-54　对广播对象去持久化

```
def unpersist(id: Long, removeFromDriver: Boolean, blocking: Boolean): Unit = {
  logDebug(s"Unpersisting TorrentBroadcast $id")
  SparkEnv.get.blockManager.master.removeBroadcast(id, removeFromDriver, blocking)
}
```

根据代码清单 5-54，可以看到 TorrentBroadcast 的 unpersist 方法实际调用了 BlockManager 的子组件 BlockManagerMaster 的 removeBroadcast 方法来实现对广播对象去持久化，有关 BlockManagerMaster 的具体介绍请参阅 6.8 节。

5.6　map 任务输出跟踪器

mapOutputTracker 用于跟踪 map 任务的输出状态，此状态便于 reduce 任务定位 map 输出结果所在的节点地址，进而获取中间输出结果。每个 map 任务或者 reduce 任务都会有其唯一标识，分别为 mapId 和 reduceId。每个 reduce 任务的输入可能是多个 map 任务

的输出，reduce 会到各个 map 任务所在的节点上拉取 Block，这一过程叫做 Shuffle。每次 Shuffle 都有唯一的标识 shuffleId。

在开始介绍 mapOutputTracker 之前，我们先来介绍 SparkEnv 中用于注册 RpcEndpoint 或者查找 RpcEndpoint 的方法 registerOrLookupEndpoint，如代码清单 5-55 所示。

代码清单5-55　注册或查找EndPoint

```
def registerOrLookupEndpoint(
    name: String, endpointCreator: => RpcEndpoint):
  RpcEndpointRef = {
  if (isDriver) {
    logInfo("Registering " + name)
    rpcEnv.setupEndpoint(name, endpointCreator)
  } else {
    RpcUtils.makeDriverRef(name, conf, rpcEnv)
  }
}
```

根据代码清单 5-55，如果当前实例是 Driver，则调用代码清单 5-46 中介绍的 setupEndpoint 方法向 Dispatcher 注册 Endpoint。如果是 Executor，则调用工具类 RpcUtils 的 makeDriverRef 方法向远端的 NettyRpcEnv 询问获取相关 RpcEndpoint 的 RpcEndpointRef。

SparkEnv 中创建 mapOutputTracker 的代码如下。

```
val mapOutputTracker = if (isDriver) {
  new MapOutputTrackerMaster(conf, broadcastManager, isLocal)
} else {
  new MapOutputTrackerWorker(conf)
}

mapOutputTracker.trackerEndpoint = registerOrLookupEndpoint(MapOutputTracker.
    ENDPOINT_NAME,
  new MapOutputTrackerMasterEndpoint(
    rpcEnv, mapOutputTracker.asInstanceOf[MapOutputTrackerMaster], conf))
```

可以看到针对当前实例是 Driver 还是 Executor，创建 mapOutputTracker 的方式有所不同。

- 如果当前应用程序是 Driver，则创建 MapOutputTrackerMaster，然后创建 MapOutputTrackerMasterEndpoint，并且注册到 Dispatcher 中，注册名为 MapOutputTracker。
- 如果当前应用程序是 Executor，则创建 MapOutputTrackerWorker，并从远端 Driver 实例的 NettyRpcEnv 的 Dispatcher 中查找 MapOutputTrackerMasterEndpoint 的引用。

无论是 Driver 还是 Executor，最后都由 mapOutputTracker 的属性 trackerEndpoint 持有 MapOutputTrackerMasterEndpoint 的引用。

小贴士：在 Spark 1.x.x 版本中存在 MapOutputTrackerMasterActor，MapOutputTrackerMasterEndpoint 如今替代了前者。

5.6.1　MapOutputTracker 的实现

无论是 MapOutputTrackerMaster 还是 MapOutputTrackerWorker，它们都继承自抽象类 MapOutputTracker。MapOutputTracker 内部定义了任务输出跟踪器的规范，所以我们先来了解它的属性，然后再看看其提供的功能。

MapOutputTracker 中的属性包括以下几项。

- trackerEndpoint：用于持有 Driver 上 MapOutputTrackerMasterEndpoint 的 RpcEndpoint-Ref。
- mapStatuses：用于维护各个 map 任务的输出状态。类型为 Map[Int, Array[MapStatus]]。其中 key 对应 shuffleId，Array 存储各个 map 任务对应的状态信息 MapStatus。由于各个 MapOutputTrackerWorker 会向 MapOutputTrackerMaster 不断汇报 map 任务的状态信息，因此 MapOutputTrackerMaster 的 mapStatuses 中维护的信息是最新最全的。MapOutputTrackerWorker 的 mapStatuses 对于本节点 Executor 运行的 map 任务状态是及时更新的，而对于其他节点上的 map 任务状态则更像一个缓存，在 mapStatuses 不能命中时会向 Driver 上的 MapOutputTrackerMaster 获取最新的任务状态信息。
- epoch：用于 Executor 故障转移的同步标记。每个 Executor 在运行的时候会更新 epoch，潜在的附加动作将清空缓存。当 Executor 丢失后增加 epoch。
- epochLock：用于保证 epoch 变量的线程安全性。
- fetching：shuffle 获取集合。数据类型为 HashSet[Int]，用来记录当前 Executor 正在从哪些 map 输出的位置拉取数据。

了解了 MapOutputTracker 中的属性，来看看 MapOutputTracker 所提供的方法。

1. 询问方法 askTracker

askTracker 方法（见代码清单 5-56）用于向 MapOutputTrackerMasterEndpoint 发送消息，并期望在超时时间之内得到回复。

代码清单5-56　askTracker的实现

```
protected def askTracker[T: ClassTag](message: Any): T = {
  try {
    trackerEndpoint.askWithRetry[T](message)
  } catch {
    case e: Exception =>
      logError("Error communicating with MapOutputTracker", e)
      throw new SparkException("Error communicating with MapOutputTracker", e)
  }
}
```

根据代码清单 5-56，askTracker 通过调用 NettyRpcEndpointRef 的 askWithRetry 方法（在 5.3.2 节已详细介绍）实现。

书籍是人类进步的阶梯

2. 发送方法 sendTracker

sendTracker 方法（见代码清单 5-57）用于向 MapOutputTrackerMasterEndpoint 发送消息，并期望在超时时间之内获得的返回值为 true。

代码清单5-57 sendTracker的实现

```
protected def sendTracker(message: Any) {
  val response = askTracker[Boolean](message)
  if (response != true) {
    throw new SparkException(
      "Error reply received from MapOutputTracker. Expecting true, got " +
          response.toString)
  }
}
```

3. getStatuses

getStatuses 方法可以根据 shuffleId 获取 MapStatus（即 map 状态信息）的数组，其实现如代码清单 5-58 所示。

代码清单5-58 getStatuses的实现

```
private def getStatuses(shuffleId: Int): Array[MapStatus] = {
  val statuses = mapStatuses.get(shuffleId).orNull
  if (statuses == null) {
    logInfo("Don't have map outputs for shuffle " + shuffleId + ", fetching
        them")
    val startTime = System.currentTimeMillis
    var fetchedStatuses: Array[MapStatus] = null
    fetching.synchronized {
      // Someone else is fetching it; wait for them to be done
      while (fetching.contains(shuffleId)) {
        try {
          fetching.wait()
        } catch {
          case e: InterruptedException =>
        }
      }
      fetchedStatuses = mapStatuses.get(shuffleId).orNull
      if (fetchedStatuses == null) {
        // We have to do the fetch, get others to wait for us.
        fetching += shuffleId
      }
    }
    if (fetchedStatuses == null) {
      // We won the race to fetch the statuses; do so
      logInfo("Doing the fetch; tracker endpoint = " + trackerEndpoint)
      // This try-finally prevents hangs due to timeouts:
      try {
        val fetchedBytes = askTracker[Array[Byte]](GetMapOutputStatuses(shuffleId))
        fetchedStatuses = MapOutputTracker.deserializeMapStatuses(fetchedBytes)
        logInfo("Got the output locations")
        mapStatuses.put(shuffleId, fetchedStatuses)
      } finally {
```

```
          fetching.synchronized {
            fetching -= shuffleId
            fetching.notifyAll()
          }
        }
      }
      logDebug(s"Fetching map output statuses for shuffle $shuffleId took " +
        s"${System.currentTimeMillis - startTime} ms")
      if (fetchedStatuses != null) {
        return fetchedStatuses
      } else {
        logError("Missing all output locations for shuffle " + shuffleId)
        throw new MetadataFetchFailedException(
          shuffleId, -1, "Missing all output locations for shuffle " + shuffleId)
      }
    } else {
      return statuses
    }
  }
```

根据代码清单 5-58，getStatuses 的执行步骤如下。

1）从当前 MapOutputTracker 的 mapStatuses 缓存中获取 MapStatus 数组，若没有就进入第 2）步，否则直接返回得到的 MapStatus 数组。

2）如果 shuffle 获取集合（即 fetching）中已经存在要取的 shuffleId（这说明已经有其他线程对此 shuffleId 的数据进行远程拉取了），那么就等待其他线程获取。等待会一直持续，直到 fetching 中不存在要取的 shuffleId（这说明其他线程对此 shuffleId 的数据进行远程拉取的操作已经结束），并再次从 mapStatuses 缓存中获取 MapStatus 数组，此时如果获取到 MapStatus 数组，则继续第 5）步。

3）如果 fetching 中不存在要取的 shuffleId，那么当前线程需要将 shuffleId 加入 fetching，以表示已经有线程对此 shuffleId 的数据进行远程拉取了。

4）调用 askTracker 方法（见代码清单 5-56）向 MapOutputTrackerMasterEndpoint 发送 GetMapOutputStatuses 消息，以获取 map 任务的状态信息。MapOutputTrackerMasterEndpoint 接收到 GetMapOutputStatuses 消息后，将 GetMapOutputStatuses 消息转换为 GetMapOutputMessage 消息，再放入 mapOutputRequests 队列，如代码清单 5-67、代码清单 5-68 所示。异步线程 MessageLoop 从队列中取出 GetMapOutputMessage，将请求的 map 任务状态信息序列化后返回给请求方，如代码清单 5-64 所示。请求方接收到 map 任务状态信息后，调用 MapOutputTracker 的 deserializeMapStatuses 方法对 map 任务状态进行反序列化操作，然后放入本地的 mapStatuses 缓存中。这次拉取可能成功，也可能失败，所以拉取结束后无论如何需要将 shuffleId 从 fetching 中移除，并唤醒那些在 fetching 的锁上等待的线程，以便这些线程能够获取自己需要的 MapStatus 数组。如果拉取成功则继续第 5）步。

5）返回得到的 MapStatus 数组。

书籍是人类进步的阶梯


注意 由于多个线程可能会并发访问 fetching，因此使用 fetching 的锁进行同步控制，以确保程序在并发下的安全性。

4. getMapSizesByExecutorId

getMapSizesByExecutorId 方法可通过 shuffleId 和 reduceId 获取存储了 reduce 所需的 map 中间输出结果的 BlockManager 的 BlockManagerId，以及 map 中间输出结果每个 Block 块的 BlockId 与大小，其实现如代码清单 5-59 所示。

代码清单5-59 getMapSizesByExecutorId的实现

```
def getMapSizesByExecutorId(shuffleId: Int, reduceId: Int)
    : Seq[(BlockManagerId, Seq[(BlockId, Long)])] = {
  getMapSizesByExecutorId(shuffleId, reduceId, reduceId + 1)
}
def getMapSizesByExecutorId(shuffleId: Int, startPartition: Int, endPartition:
   Int)
    : Seq[(BlockManagerId, Seq[(BlockId, Long)])] = {
  logDebug(s"Fetching outputs for shuffle $shuffleId, partitions $startPartition-
     $endPartition")
  val statuses = getStatuses(shuffleId)
  statuses.synchronized {
    return MapOutputTracker.convertMapStatuses(shuffleId, startPartition,
       endPartition, statuses)
  }
}
```

代码清单 5-59 中，MapOutputTracker 的 convertMapStatuses 方法用于将 Array[MapStatus] 转换为 Seq[(BlockManagerId, Seq[(BlockId, Long)])]，其具体实现比较简单，感兴趣的读者可自己查阅。

5. getStatistics

getStatistics 用于获取 shuffle 依赖的各个 map 输出 Block 大小的统计信息，其实现如代码清单 5-60 所示。

代码清单5-60 获取MapOutputStatistics

```
def getStatistics(dep: ShuffleDependency[_, _, _]): MapOutputStatistics = {
  val statuses = getStatuses(dep.shuffleId)
  statuses.synchronized {
    val totalSizes = new Array[Long](dep.partitioner.numPartitions)
    for (s <- statuses) {
      for (i <- 0 until totalSizes.length) {
        totalSizes(i) += s.getSizeForBlock(i)
      }
    }
    new MapOutputStatistics(dep.shuffleId, totalSizes)
  }
}
```

书籍是人类进步的阶梯

6. updateEpoch

当 Executor 运行出现故障时，Master 会再分配其他 Executor 运行任务，此时会调用 updateEpoch 方法（见代码清单 5-61）更新纪元，并且清空 mapStatuses。

代码清单5-61　更新Executor的纪元

```
def updateEpoch(newEpoch: Long) {
  epochLock.synchronized {
    if (newEpoch > epoch) {
      logInfo("Updating epoch to " + newEpoch + " and clearing cache")
      epoch = newEpoch
      mapStatuses.clear()
    }
  }
}
```

7. unregisterShuffle

unregisterShuffle 方法用于 ContextCleaner 清除 shuffleId 对应 MapStatus 的信息，其实现如代码清单 5-62 所示。

代码清单5-62　清除shuffleId对应MapStatus的信息

```
def unregisterShuffle(shuffleId: Int) {
  mapStatuses.remove(shuffleId)
}
```

5.6.2　MapOutputTrackerMaster 的实现原理

通常而言，我们所说的 mapOutputTracker 都是指 MapOutputTrackerMaster，而不是 MapOutputTrackerWorker。MapOutputTrackerWorker 将 map 任务的跟踪信息，通过 MapOutputTrackerMasterEndpoint 的 RpcEndpointRef 发送给 MapOutputTrackerMaster，由 MapOutput TrackerMaster 负责整理和维护所有的 map 任务的输出跟踪信息。MapOutput-TrackerMasterEndpoint 位于 MapOutputTrackerMaster 内部，二者只存在于 Driver 上。有了对 MapOutputTrackerMaster 的基本认识，下面详细分析它的实现。

为了便于理解 MapOutputTrackerMaster 的实现原理，首先了解它的各个属性，将使我们受益良多。MapOutputTrackerMaster 的属性包括以下几项。

- cacheEpoch：对 MapOutputTracker 的 epoch 的缓存。
- minSizeForBroadcast：用于广播的最小大小。可以使用 spark.shuffle.mapOutput.minSizeForBroadcast 属性配置，默认为 512KB。minSizeForBroadcast 必须小于 maxRpcMessageSize。
- shuffleLocalityEnabled：是否为 reduce 任务计算本地性的偏好。可以使用 spark.shuffle.reduceLocality.enabled 属性进行配置，默认为 true。
- mapStatuses：用于存储 shuffleId 与 Array[MapStatus] 的映射关系。由于 MapStatus

书籍是人类进步的阶梯

维护了 map 输出 Block 的地址 BlockManagerId，所以 reduce 任务知道从何处获取 map 任务的中间输出。在 6.1.2 节对 BlockManagerId 有更加详细的介绍。

- cachedSerializedStatuses：用于存储 shuffleId 与序列化后的状态的映射关系。其中 key 对应 shuffleId，value 为对 MapStatus 序列化后的字节数组。
- maxRpcMessageSize：最大的 Rpc 消息的大小。此属性可以通过 spark.rpc.message.maxSize 属性进行配置，默认为 128，单位是 MB。minSizeForBroadcast 必须小于 maxRpcMessageSize。maxRpcMessageSize 实际通过调用 RpcUtils 的 maxMessageSizeBytes 方法获得，其具体实现请参阅附录 H。
- cachedSerializedBroadcast：用于缓存序列化的广播变量，保持与 cachedSerializedStatuses 的同步。当需要移除 shuffleId 在 cachedSerializedStatuses 中的状态数据时，此缓存中的数据也会被移除。
- shuffleIdLocks：每个 shuffleId 对应的锁。当 shuffle 过程开始时，会有大量的关于同一个 shuffle 的请求，使用锁可以避免对同一 shuffle 的多次序列化。
- mapOutputRequests：使用阻塞队列来缓存 GetMapOutputMessage（获取 map 任务输出）的请求。
- threadpool：用于获取 map 输出的固定大小的线程池。此线程池提交的线程都以后台线程运行，且线程名以 map-output-dispatcher 为前缀，线程池大小可以使用 spark.shuffle.mapOutput.dispatcher.numThreads 属性配置，默认大小为 8。

1. MapOutputTrackerMaster 的运行原理

在创建 MapOutputTrackerMaster 的最后，会创建对 map 输出请求进行处理的线程池 threadpool，如代码清单 5-63 所示。

代码清单5-63　创建MapOutputTrackerMaster中的线程池

```
private val threadpool: ThreadPoolExecutor = {
  val numThreads = conf.getInt("spark.shuffle.mapOutput.dispatcher.numThreads", 8)
  val pool = ThreadUtils.newDaemonFixedThreadPool(numThreads, "map-output-
    dispatcher")
  for (i <- 0 until numThreads) {
    pool.execute(new MessageLoop)
  }
  pool
}
```

根据代码清单 5-63，可以看到创建 threadpool 线程池的步骤如下。

1）获取此线程池的大小 numThreads。此线程池的大小默认为 8，也可以使用 spark.shuffle.mapOutput.dispatcher.numThreads 属性配置。

2）创建线程池。此线程池是固定大小的线程池，并且启动的线程都以后台线程方式运行，且线程名以 map-output-dispatcher 为前缀。

3）启动与线程池大小相同数量的线程，每个线程执行的任务都是 MessageLoop。

书籍是人类进步的阶梯

4）返回此线程池的引用。

上面介绍的线程池 threadpool 的创建与 5.3.4 节介绍的 Dispatcher 内的线程池 threadpool 从命名和方式上非常相似，而且此处启动的任务也是 MessageLoop。不过这里的 MessageLoop 并非是 Dispatcher 中的 MessageLoop，而是 MapOutputTrackerMaster 的内部类。MapOutputTrackerMaster 中的 MessageLoop 也实现了 Java 的 Runnable 接口，其实现如代码清单 5-64 所示。

代码清单5-64　MessageLoop的实现

```
private class MessageLoop extends Runnable {
  override def run(): Unit = {
    try {
      while (true) {
        try {
          val data = mapOutputRequests.take()
           if (data == PoisonPill) {
            // Put PoisonPill back so that other MessageLoops can see it.
            mapOutputRequests.offer(PoisonPill)
            return
          }
          val context = data.context
          val shuffleId = data.shuffleId
          val hostPort = context.senderAddress.hostPort
          logDebug("Handling request to send map output locations for shuffle "
              + shuffleId +
            " to " + hostPort)
          val mapOutputStatuses = getSerializedMapOutputStatuses(shuffleId)
          context.reply(mapOutputStatuses)
        } catch {
          case NonFatal(e) => logError(e.getMessage, e)
        }
      }
    } catch {
      case ie: InterruptedException => // exit
    }
  }
}
```

根据代码清单 5-64，我们看到 MessageLoop 果如其名，在循环过程中不断对新的消息进行处理。每次循环中的逻辑如下。

1）从 mapOutputRequests 中获取 GetMapOutputMessage。由于 mapOutputRequests 是个阻塞队列，所以当 mapOutputRequests 中没有 GetMapOutputMessage 时，MessageLoop 线程会被阻塞。GetMapOutputMessage 是个样例类，包含了 shuffleId 和 RpcCallContext 两个属性，RpcCallContext 用于服务端回复客户端，其实现如下。

```
private[spark] case class GetMapOutputMessage(shuffleId: Int, context: RpcCallContext)
```

2）如果取到的 GetMapOutputMessage 是 PoisonPill（“毒药”），那么此 MessageLoop 线程将退出（通过 return 语句）。这里有个动作，就是将 PoisonPill 重新放入到 mapOutput-

书籍是人类进步的阶梯

Requests 中，这是因为 threadpool 线程池极有可能不止一个 MessageLoop 线程，为了让大家都“毒发身亡”，还需要把“毒药”放回到 receivers 中，这样其他“活着”的线程就会再次误食“毒药”，达到所有 MessageLoop 线程都结束的效果。

3）如果取到的 GetMapOutputMessage 不是“毒药”，那么调用 getSerializedMapOutputStatuses 方法获取 GetMapOutputMessage 携带的 shuffleId 所对应的序列化任务状态信息。

4）调用 RpcCallContext 的回调方法 reply，将序列化的 map 任务状态信息返回给客户端（即其他节点的 Executor）。

MessageLoop 任务执行的第 3）步调用了 getSerializedMapOutputStatuses 方法获取序列化任务状态信息，其具体实现如代码清单 5-65 所示。

代码清单5-65 获取序列化任务状态信息

```
def getSerializedMapOutputStatuses(shuffleId: Int): Array[Byte] = {
  var statuses: Array[MapStatus] = null
  var retBytes: Array[Byte] = null
  var epochGotten: Long = -1
    // 省略次要代码
  if (checkCachedStatuses()) return retBytes
  var shuffleIdLock = shuffleIdLocks.get(shuffleId)
  if (null == shuffleIdLock) {
    val newLock = new Object()
    val prevLock = shuffleIdLocks.putIfAbsent(shuffleId, newLock)
    shuffleIdLock = if (null != prevLock) prevLock else newLock
  }
  shuffleIdLock.synchronized {
    if (checkCachedStatuses()) return retBytes
    val (bytes, bcast) = MapOutputTracker.serializeMapStatuses(statuses,
       broadcastManager,
      isLocal, minSizeForBroadcast)
    logInfo("Size of output statuses for shuffle %d is %d bytes".format (shuffleId,
        bytes.length))
    // Add them into the table only if the epoch hasn't changed while we were
        working
    epochLock.synchronized {
      if (epoch == epochGotten) {
        cachedSerializedStatuses(shuffleId) = bytes
        if (null != bcast) cachedSerializedBroadcast(shuffleId) = bcast
      } else {
        logInfo("Epoch changed, not caching!")
        removeBroadcast(bcast)
      }
    }
    bytes
  }
}
```

根据代码清单 5-65，getSerializedMapOutputStatuses 方法的执行步骤如下。

1）调用 checkCachedStatuses 检查 cachedSerializedStatuses 中是否有 shuffleId 所对应的序列化的任务状态信息。如果有，则从 cachedSerializedStatuses 中获取序列化的任务状

书籍是人类进步的阶梯

态信息后返回，否则进入第 2）步。

2）从 shuffleIdLocks 中获取 shuffleId 对应的锁，如果没有，则新建一个对象作为锁放入 shuffleIdLocks 中。

3）获取 mapStatuses 中缓存的 MapStatus 数组，然后调用 MapOutputTracker 的 serializeMapStatuses 方法将 MapStatus 数组序列化，最后调用 BroadcastManager 的 newBroadcast 方法（详见 5.5 节）将序列化 MapStatus 数组后产生的字节数组进行广播。serializeMapStatuses 方法将返回序列化的 MapStatus（即 bytes）和广播对象 bcast 的对偶。

4）如果当前 epoch 没有发生变化，将 shuffleId 与序列化的 MapStatus 之间的映射关系放入 cachedSerializedStatuses，并将 shuffleId 与广播对象 bcast 之间的映射关系放入 cachedSerializedBroadcast。

5）如果当前 epoch 已经发生变化，则不需要对 bytes 和 bcast 进行缓存，同时调用 removeBroadcast 方法删除广播对象 bcast。

getSerializedMapOutputStatuses 方法中调用了 removeBroadcast 方法移除广播对象。removeBroadcast 的实现如代码清单 5-66 所示。

代码清单5-66　移除广播对象

```
private def removeBroadcast(bcast: Broadcast[]): Unit = {
  if (null != bcast) {
    broadcastManager.unbroadcast(bcast.id,
      removeFromDriver = true, blocking = false)
  }
}
```

根据代码清单 5-66，removeBroadcast 方法的功能实际委托给 BroadcastManager 的 unbroadcast 方法（在 5.5 节有介绍）处理了。

MapOutputTrackerMaster 中的 MessageLoop 任务在不断地消费着阻塞队列 mapOutputRequests 中的 GetMapOutputMessage，那么 GetMapOutputMessage 的来源在哪里？GetMapOutputMessage 又是如何放入 mapOutputRequests 中的呢？在 MapOutputTrackerMaster 中要将 GetMapOutputMessage 放入 mapOutputRequests，主要通过调用 MapOutputTrackerMaster 的 post 方法，其实现如代码清单 5-67 所示。

代码清单5-67　投递map输出结果获取请求

```
def post(message: GetMapOutputMessage): Unit = {
  mapOutputRequests.offer(message)
}
```

MapOutputTrackerMasterEndpoint 组件用于接收获取 map 中间状态和停止对 map 中间状态进行跟踪的请求，它实现了特质 RpcEndpoint 并重写了 receiveAndReply 方法，如代码清单 5-68 所示。

书籍是人类进步的阶梯

代码清单5-68 MapOutputTrackerMasterEndpoint的定义

```
private[spark] class MapOutputTrackerMasterEndpoint(
    override val rpcEnv: RpcEnv, tracker: MapOutputTrackerMaster, conf: SparkConf)
  extends RpcEndpoint with Logging {
// 省略次要代码
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
    case GetMapOutputStatuses(shuffleId: Int) =>
      val hostPort = context.senderAddress.hostPort
      logInfo("Asked to send map output locations for shuffle " + shuffleId + "
          to " + hostPort)
      val mapOutputStatuses = tracker.post(new GetMapOutputMessage(shuffleId,
         context))
  case StopMapOutputTracker =>
    logInfo("MapOutputTrackerMasterEndpoint stopped!")
    context.reply(true)
    stop()
  }
}
```

根据代码清单5-68，可以看到MapOutputTrackerMasterEndpoint将接收两种类型的请求。

- **获取map中间输出状态**。当接收到GetMapOutputStatuses消息（主要由MapOutputTracker的getStatuses方法调用askTracker时产生，如代码清单5-58所示）后，将调用MapOutputTrackerMaster的post方法投递GetMapOutputMessage类型的消息。
- **停止map中间输出的跟踪**。首先回调RpcCallContext的reply方法向客户端返回true，然后调用MapOutputTrackerMaster的stop方法（见代码清单5-69）停止MapOutputTrackerMaster。

代码清单5-69 停止MapOutputTrackerMaster

```
override def stop() {
  mapOutputRequests.offer(PoisonPill)
  threadpool.shutdown()
  sendTracker(StopMapOutputTracker)
  mapStatuses.clear()
  trackerEndpoint = null
  cachedSerializedStatuses.clear()
  clearCachedBroadcast()
  shuffleIdLocks.clear()
}
```

根据代码清单5-69，可以看到stop方法的执行步骤如下。

1）向mapOutputRequests投递PoisonPill（“毒药”）。此操作会向mapOutputRequests队列中添加PoisonPill，进而停止MapOutputTrackerMaster中的所有MessageLoop线程。

2）关闭threadpool。由于MapOutputTrackerMaster中的所有MessageLoop线程都已经停止，因而可以平滑关闭threadpool。

书籍是人类进步的阶梯

3）调用 sendTracker 向 MapOutputTrackerMasterEndpoint 发送 StopMapOutputTracker 消息。这里有个小问题，那就是 MapOutputTrackerMasterEndpoint 再次收到 StopMapOutputTracker 消息，进而引起 stop 方法的循环调用。不知道这是否会引起问题。

4）清空 mapStatuses。

5）回收 MapOutputTrackerMasterEndpoint。

6）清空 cachedSerializedStatuses。

7）清空 shuffleIdLocks。

有了以上的一系列分析，现在我们可以用图 5-13 展现 MapOutputTrackerMaster 的运行原理。

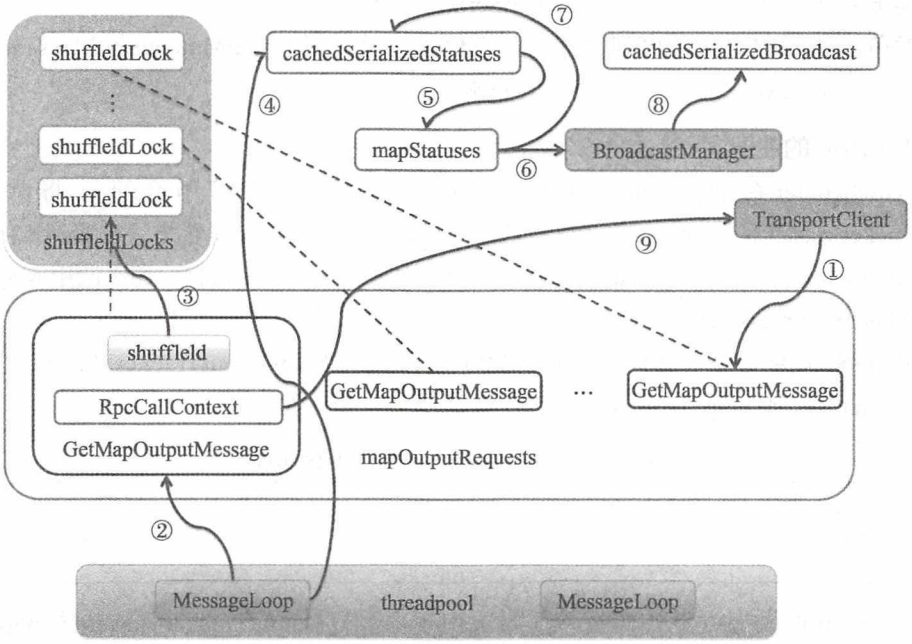


图 5-13　MapOutputTrackerMaster 的运行原理简图

这里对图 5-13 的内容再做一次简短描述。

序号①：表示某个 Executor 调用 MapOutputTrackerWorker 的 getStatuses 方法获取某个 shuffle 的 map 任务状态信息，当发现本地的 mapStatuses 没有相应的缓存，则调用 askTracker 方法发送 GetMapOutputStatuses 消息。根据代码清单 5-56 的内容我们知道，askTracker 实际是通过 MapOutputTrackerMasterEndpoint 的 NettyRpcEndpointRef 向远端发送 GetMapOutputStatuses 消息。发送实际依托于 NettyRpcEndpointRef 持有的 Transport-Client。MapOutputTrackerMasterEndpoint 在接收到 GetMapOutputStatuses 消息后，将 GetMapOutputMessage 消息放入 mapOutput Requests 队尾。

序号②：表示 MessageLoop 线程从 mapOutputRequests 队头取出 GetMapOutputMessage。

书籍是人类进步的阶梯

序号③：表示从 shuffleIdLocks 数组中取出与当前 GetMapOutputMessage 携带的 shuffleId 相对应的锁。

序号④：表示首先从 cachedSerializedStatuses 缓存中获取 shuffleId 对应的序列化任务状态信息。

序号⑤：表示当 cachedSerializedStatuses 中没有 shuffleId 对应的序列化任务状态信息，则获取 mapStatuses 中缓存的 shuffleId 对应的任务状态数组。

序号⑥：表示将任务状态数组进行序列化，然后使用 BroadcastManager 对序列化的任务状态进行广播。

序号⑦：表示将序列化的任务状态放入 cachedSerializedStatuses 缓存中。

序号⑧：表示将广播对象放入 cachedSerializedBroadcast 缓存中。

序号⑨：表示将获得的序列化任务状态信息，通过回调 GetMapOutputMessage 消息携带的 RpcCallContext 的 reply 方法回复客户端。

2. Shuffle 的注册

DAGScheduler 在创建 ShuffleMapStage 的时候（将在第 7.4.7 节介绍），将调用 MapOutputTrackerMaster 的 containsShuffle 方法（见代码清单 5-70），查看是否已经存在 shuffleId 对应的 MapStatus。如果 MapOutputTrackerMaster 中未注册此 shuffleId，那么调用 MapOutputTrackerMaster 的 registerShuffle 方法（见代码清单 5-71）注册 shuffleId。

containsShuffle 方法（见代码清单 5-70）用于查找 MapOutputTrackerMaster 的 cachedSerializedStatuses 或 mapStatuses 中是否已经注册了 shuffleId。

代码清单5-70 containsShuffle的实现

```
def containsShuffle(shuffleId: Int): Boolean = {
  cachedSerializedStatuses.contains(shuffleId) || mapStatuses.contains(shuffleId)
}
```

registerShuffle 方法（见代码清单 5-71）用于向 MapOutputTrackerMaster 的 mapStatuses 中注册 shuffleId 与对应的 MapStatus 的映射关系。

代码清单5-71 registerShuffle的实现

```
def registerShuffle(shuffleId: Int, numMaps: Int) {
  if (mapStatuses.put(shuffleId, new Array[MapStatus](numMaps)).isDefined) {
    throw new IllegalArgumentException("Shuffle ID " + shuffleId + " registered
        twice")
  }
  shuffleIdLocks.putIfAbsent(shuffleId, new Object())
}
```

根据代码清单 5-71，可以看到注册 shuffleId 时，shuffleId 在 mapStatuses 中对应的是以 map 任务数量作为长度的数组，此数组将用于保存 MapStatus。注册了 Shuffle，那么 MapStatus 是何时保存的呢？当 ShuffleMapStage 内的所有 ShuffleMapTask 运行成功后，将

调用 MapOutputTrackerMaster 的 registerMapOutputs 方法（ShuffleMapTask 运行结果的处理将在 7.4.11 节介绍）。registerMapOutputs 方法的实现如代码清单 5-72 所示。

代码清单5-72　registerMapOutputs方法

```
def registerMapOutputs(shuffleId: Int, statuses: Array[MapStatus], changeEpoch:
    Boolean = false)
{
  mapStatuses.put(shuffleId, statuses.clone())
  if (changeEpoch) {
    incrementEpoch()
  }
}
```

根据代码清单 5-72，registerMapOutputs 方法将把 ShuffleMapStage 中每个 ShuffleMapTask 的 MapStatus 保存到 shuffleId 在 mapStatuses 中对应的数组中。

整个 Shuffle 的注册流程可以用图 5-14 来表示。

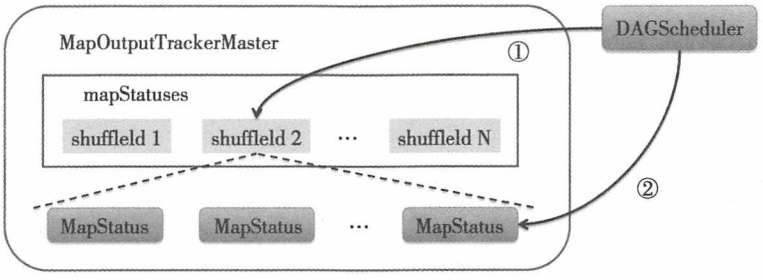


图 5-14　Shuffle 的注册流程

其处理步骤包括以下两步。

序号①：表示 DAGScheduler 在创建了 ShuffleMapStage 后，调用 MapOutputTrackerMaster 的 registerShuffle 方法向 mapStatuses 缓存注册 shuffleid。

序号②：表示 DAGScheduler 处理 ShuffleMapTask 的执行结果时，如果发现 ShuffleMapTask 所属的 ShuffleMapStage 中每一个分区的 ShuffleMapTask 都执行成功了，那么将调用 MapOutputTrackerMaster 的 registerMapOutputs 方法，将 ShuffleMapStage 中每一个 ShuffleMapTask 的 MapStatus 保存到 shuffleid 对应的 MapStatus 数组中。

5.7　构建存储体系

早在介绍 BroadcastManager 的时候，我们就知道 BroadcastManager 的底层都依赖于 SparkEnv 的存储体系。存储体系中最重要的组件包括 Shuffle 管理器 ShuffleManager、内存管理器 MemoryManager、块传输服务 BlockTransferService、对所有 BlockManager 进行管理的 BlockManagerMaster、磁盘块管理器 DiskBlockManager、块锁管理器 BlockInfoManager 及

块管理器 BlockManager。存储体系的内容将在第 6 章详细介绍，本节只简单介绍 SparkEnv 中存储体系各个组件的实例化，如代码清单 5-73 所示。

代码清单5-73 构建存储体系

```
val shortShuffleMgrNames = Map(
  "sort" -> classOf[org.apache.spark.shuffle.sort.SortShuffleManager].getName,
  "tungsten-sort" -> classOf[org.apache.spark.shuffle.sort.SortShuffleManager].
      getName)
val shuffleMgrName = conf.get("spark.shuffle.manager", "sort")
val shuffleMgrClass = shortShuffleMgrNames.getOrElse(shuffleMgrName.toLowerCase,
   shuffleMgrName)
val shuffleManager = instantiateClass[ShuffleManager](shuffleMgrClass)
val useLegacyMemoryManager = conf.getBoolean("spark.memory.useLegacyMode", false)
val memoryManager: MemoryManager =
  if (useLegacyMemoryManager) {
    new StaticMemoryManager(conf, numUsableCores)
  } else {
    UnifiedMemoryManager(conf, numUsableCores)
  }
val blockManagerPort = if (isDriver) {
  conf.get(DRIVER_BLOCK_MANAGER_PORT)
} else {
  conf.get(BLOCK_MANAGER_PORT)
}
val blockTransferService =
    new NettyBlockTransferService(conf, securityManager, bindAddress, advertiseAddress,
      blockManagerPort, numUsableCores)
val blockManagerMaster = new BlockManagerMaster(registerOrLookupEndpoint(
    BlockManagerMaster.DRIVER_ENDPOINT_NAME,
    new BlockManagerMasterEndpoint(rpcEnv, isLocal, conf, listenerBus)),
    conf, isDriver)
val blockManager = new BlockManager(executorId, rpcEnv, blockManagerMaster,
    serializerManager, conf, memoryManager, mapOutputTracker, shuffleManager,
    blockTransferService, securityManager, numUsableCores)
```

根据代码清单 5-73，我们知道 SparkEnv 中存储体系各个组件的实例化步骤如下。

1）根据 spark.shuffle.manager 属性，实例化 ShuffleManager。Spark 2.x.x 版本提供了 sort 和 tungsten-sort 两种 ShuffleManager 的实现。无论是 sort 还是 tungsten-sort，我们看到其实现类都是 SortShuffleManager。本书将在第 8 章详细介绍 SortShuffleManager。

2）MemoryManager 的主要实现有 StaticMemoryManager 和 UnifiedMemoryManager。StaticMemoryManager 是 Spark 早期版本遗留下来的内存管理器实现，可以配置 spark.memory.useLegacyMode 属性来指定，该属性默认为 false，因此默认的内存管理器是 UnifiedMemoryManager。

3）获取当前 SparkEnv 的块传输服务 BlockTransferService 对外提供的端口号。如果当前实例是 Driver，则从 SparkConf 中获取由常量 DRIVER_BLOCK_MANAGER_PORT 指定的端口。如果当前实例是 Executor，则从 SparkConf 中获取由常量 BLOCK_MANAGER_PORT 指定的端口。这两个常量的定义如下。

```
private[spark] val BLOCK_MANAGER_PORT = ConfigBuilder("spark.blockManager.port")
  .doc("Port to use for the block manager when a more specific setting is not
     provided.")
  .intConf
  .createWithDefault(0)
private[spark] val DRIVER_BLOCK_MANAGER_PORT = ConfigBuilder("spark.driver.
    blockManager.port")
  .doc("Port to use for the block manager on the driver.")
  .fallbackConf(BLOCK_MANAGER_PORT)
```

因此我们知道，可以通过指定 spark.driver.blockManager.port 属性或者 spark.blockManager.port 属性对 BlockTransferService 的端口进行配置。

4）创建块传输服务 BlockTransferService。这里使用的是 BlockTransferService 的子类 NettyBlockTransferService，NettyBlockTransferService 将提供对外的块传输服务。也正是因为 MapOutputTracker 与 NettyBlockTransferService 的配合，才实现了 Spark 的 Shuffle。

5）查找或注册 BlockManagerMasterEndpoint。这里的 registerOrLookupEndpoint 方法（见代码清单 5-55）已在 5.6 节介绍过，因此：

① 当前应用程序是 Driver，则创建 BlockManagerMasterEndpoint，并且注册到 Dispatcher 中，注册名为 BlockManagerMaster；

② 当前应用程序是 Executor，则从远端 Driver 实例的 NettyRpcEnv 的 Dispatcher 中查找 BlockManagerMasterEndpoint 的引用。

无论是 Driver 还是 Executor，最后都由 BlockManagerMaster 的属性 driverEndpoint 持有 BlockManagerMasterEndpoint 的引用。

小贴士：在 Spark 1.x.x 版本中存在 BlockManagerMasterActor，BlockManagerMasterEndpoint 如今替代了前者。

6）创建 BlockManagerMaster。

7）创建 BlockManager。此处只创建了 BlockManager，只有其 init 方法被调用后，BlockManager 才能正常工作。

5.8　创建度量系统

在 3.4 节，我们详细介绍了度量系统的基本架构与概念。在 SparkEnv 中，度量系统也是必不可少的一个子组件。因此本节将详细介绍 SparkEnv 中度量系统的创建，如代码清单 5-74 所示。

代码清单5-74　SparkEnv中度量系统的创建

```
val metricsSystem = if (isDriver) {
  MetricsSystem.createMetricsSystem("driver", conf, securityManager)
} else {
```

```
    conf.set("spark.executor.id", executorId)
    val ms = MetricsSystem.createMetricsSystem("executor", conf, securityManager)
    ms.start()
    ms
  }
```

根据代码清单 5-74，我们知道创建度量系统根据当前实例是 Driver 还是 Executor 也有区别。

- 当前实例为 Driver：创建度量系统，并且制定度量系统的实例名为 driver。此时虽然创建了，但是并未启动，目的是等待 SparkContext 中的任务调度器 Task Scheculer 告诉度量系统应用程序 ID 后再启动。
- 当前实例为 Executor：设置 spark.executor.id 属性为当前 Executor 的 ID，然后创建并启动度量系统（启动 MetricsSystem 的内容详见 5.8.3 节）。

创建度量系统使用了伴生对象 MetricsSystem 的 createMetricsSystem 方法（类似于 Java 的静态方法），其实现如代码清单 5-75 所示。

代码清单5-75 createMetricsSystem的实现

```
def createMetricsSystem(
    instance: String, conf: SparkConf, securityMgr: SecurityManager):
        MetricsSystem = {
  new MetricsSystem(instance, conf, securityMgr)
}
```

根据代码清单 5-75，createMetricsSystem 方法实际利用了 MetricsSystem 的构造器来创建度量系统。

要理解 MetricsSystem 的实现原理，首先应当从了解它的成员属性开始，MetricsSystem 拥有的成员属性如下。

- instance：度量系统的实例名。例如，Master、Worker、Application、Driver 及 Executor 等。
- metricsConfig：度量配置。metricsConfig 的类型为 MetricsConfig，主要提供对度量配置的设置、加载、转换等功能。MetricsConfig 中的度量配置包括了 Sink 和 Source，因此 MetricsSystem 将根据 MetricsConfig 构建度量系统的所有 Sink 和 Source。
- sinks：Sink 的数组。在 3.4.2 节介绍过 Sink（即度量输出）的设计与实现。sinks 用于缓存所有注册到 MetricsSystem 的度量输出。
- sources：Source 的数组。在 3.4.1 节介绍过 Source（即度量的采集或来源）的设计与实现。sources 用于缓存所有注册到 MetricsSystem 的 Source。
- registry：度量注册点 MetricRegistry。Source 和 Sink 实际都是通过 MetricRegistry 注册到 Metrics 的度量仓库中的。Metrics 是 codahale 提供的第三方度量仓库，这里的 MetricRegistry 是 Metrics 提供的 API，对于 Metrics 及 MetricRegistry 的详细内

容请参阅附录 D。

- running：用于标记当前 MetricsSystem 是否正在运行。
- metricsServlet：metricsServlet 将在添加 ServletContextHandler 后通过 Web UI 展示。metricsServlet 的类型是 Option[MetricsServlet]。

下面将逐一介绍 MetricsSystem 内的这些成员。

5.8.1　MetricsConfig 详解

MetricsConfig 保存了 MetricsSystem 的配置信息，有以下两个成员属性。

- properties：度量的属性信息。类型为 Properties。
- perInstanceSubProperties：每个实例的子属性。缓存每个实例与其属性的映射关系，类型为 HashMap[String, Properties]。

有了对 MetricsConfig 中属性的了解，我们来看看 MetricsConfig 中的几个方法。

1. 设置默认属性

setDefaultProperties 方法（见代码清单 5-76）用于给 MetricsConfig 的 properties 中添加默认的度量属性。

代码清单5-76　设置默认的度量属性

```
private def setDefaultProperties(prop: Properties) {
   prop.setProperty("*.sink.servlet.class", "org.apache.spark.metrics.sink.
MetricsServlet")
  prop.setProperty("*.sink.servlet.path", "/metrics/json")
  prop.setProperty("master.sink.servlet.path", "/metrics/master/json")
  prop.setProperty("applications.sink.servlet.path", "/metrics/applications/json")
}
```

根据代码清单 5-76，MetricsConfig 一共有四个默认属性。

2. 从文件中加载度量属性

loadPropertiesFromFile 方法（见代码清单 5-77）用于给 MetricsConfig 的 properties 中添加从指定文件中加载的度量属性。

代码清单5-77　从文件中加载度量属性

```
private[this] def loadPropertiesFromFile(path: Option[String]): Unit = {
  var is: InputStream = null
  try {
    is = path match {
      case Some(f) => new FileInputStream(f)
      case None => Utils.getSparkClassLoader.getResourceAsStream(DEFAULT_METRICS_
          CONF_FILENAME)
    }
    if (is != null) {
      properties.load(is)
    }
```

```
    } catch {
      case e: Exception =>
        val file = path.getOrElse(DEFAULT_METRICS_CONF_FILENAME)
        logError(s"Error loading configuration file $file", e)
    } finally {
      if (is != null) {
        is.close()
      }
    }
  }
```

根据代码清单 5-77，我们知道如果没有指定度量属性文件或者此文件不存在，那么将从类路径下的属性文件 metrics.properties（即常量 DEFAULT_METRICS_CONF_FILENAME 的值）中加载度量属性。读取类路径下的文件时使用了工具类 Utils 的 getSparkClassLoader 方法，其具体实现可以参阅附录 A。

3. 提取实例的属性

subProperties 方法（见代码清单 5-78）对于 properties 中的每个属性 kv，通过正则表达式匹配找出 kv 的 key 的前缀和后缀，以前缀为实例，后缀作为新的属性 kv2 的 key，kv 的 value 作为新的属性 kv2 的 value，最后将属性 kv2 添加到实例对应的属性集合中作为实例的属性。这些解释过于书面，在介绍 MetricsConfig 的初始化时，将会有更直观的例子。

代码清单5-78 提取实例的属性

```
def subProperties(prop: Properties, regex: Regex): mutable.HashMap[String,
Properties] = {
  val subProperties = new mutable.HashMap[String, Properties]
  prop.asScala.foreach { kv =>
    if (regex.findPrefixOf(kv._1.toString).isDefined) {
      val regex(prefix, suffix) = kv._1.toString
      subProperties.getOrElseUpdate(prefix, new Properties).setProperty(suffix,
          kv._2.toString)
    }
  }
  subProperties
}
```

4. 初始化

在构造 MetricsSystem 的过程中将会执行以下代码。

```
metricsConfig.initialize()
```

MetricsConfig 的 initialize 方法的实现如代码清单 5-79 所示。

代码清单5-79 MetricsConfig的初始化

```
def initialize() {
  setDefaultProperties(properties)
  loadPropertiesFromFile(conf.getOption("spark.metrics.conf"))
  val prefix = "spark.metrics.conf."
```

```
    conf.getAll.foreach {
      case (k, v) if k.startsWith(prefix) =>
        properties.setProperty(k.substring(prefix.length()), v)
      case _ =>
    }

    perInstanceSubProperties = subProperties(properties, INSTANCE_REGEX)
    if (perInstanceSubProperties.contains(DEFAULT_PREFIX)) {
      val defaultSubProperties = perInstanceSubProperties(DEFAULT_PREFIX).asScala
      for ((instance, prop) <- perInstanceSubProperties if (instance != DEFAULT_
        PREFIX);
          (k, v) <- defaultSubProperties if (prop.get(k) == null)) {
        prop.put(k, v)
      }
    }
  }
```

根据代码清单 5-79，MetricsConfig 的初始化过程如下。

1）设置默认属性。setDefaultProperties 的实现如代码清单 5-76 所示。

2）从文件中加载度量属性。可以通过 spark.metrics.conf 属性指定度量属性文件。loadPropertiesFromFile 的实现如代码清单 5-77 所示。

3）从 SparkConf 中查找以 spark.metrics.conf. 为前缀的配置属性，并且截取 key 的前缀后的部分作为度量属性的 key/value 不变。

4）提取实例的属性。这里指定了正则表达式 ^(*|[a-zA-Z]+)\.(.+) 找出 properties 中各个实例的属性。subProperties 的实现如代码清单 5-78 所示。

为了更容易理解代码清单 5-78 的实现，这里以代码清单 5-76 中的 KV 对 "*.sink.servlet.class"->"org.apache.spark.metrics.sink.MetricsServlet" 和 "*.sink.servlet.path" ->"/metrics/json" 为例。首先使用正则表达式 ^(*|[a-zA-Z]+)\.(.+) 匹配出两个元组，那么第一个 KV 对的两个元组分别为 * 和 sink.servlet.class。第二个 KV 对的两个元组分别为 * 和 sink.servlet.path。然后以第一个元组作为前缀，第二个元组作为后缀。以前缀为实例，后缀为新属性的 key，KV 对的 value 作为新属性的 value。最后将新属性添加到实例对应的 Properties 中，将实例与 Properties 之间的映射关系存入 Map。结果如下。

```
Map(*->{
  sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
  sink.servlet.path=/metrics/json
})
```

根据这个例子，4 个默认度量属性提取出实例的属性后为：

```
Map(applications->{
  sink.servlet.path=/metrics/applications/json
},
master ->{
  sink.servlet.path=/metrics/master/json
},
*->{
```

```
  sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
  sink.servlet.path=/metrics/json
})
```

即实例 applications、master、* 与它们对应的属性集合。

5）向子属性中添加缺失的默认子属性（所谓默认子属性，即在 Map 中以 * 为 key 的属性）。为了理解更加简单，这里以第 4) 步得到的 Map 为例。默认子属性中的 sink.servlet.class 属性是其他两个（applications 和 master）子属性中不具有的属性，因此需要将此属性添加到以 applications 和 master 为 key 的子属性中。因此得到的结果为：

```
Map(applications->{
  sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
  sink.servlet.path=/metrics/applications/json
},
master->{
  sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
  sink.servlet.path=/metrics/master/json
},
*->{
  sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
  sink.servlet.path=/metrics/json
})
```

经过以上处理，每个度量实例与其属性集合的信息都存储在 perInstanceSubProperties 了。细心的读者可能发现在默认配置下，Spark 度量系统的属性配置只有 Sink，却没有 Source，没有了度量源，光有输出能有什么用？度量系统中的 Sink 都通过 Spark 属性或者从文件中加载，而 Source 除了与 Sink 相同的两种方式外，Spark 的实现中，很多地方都是调用 MetricsSystem 的 registerSource 方法（将在 5.8.2 节介绍）注册各种 Source 的。虽然如此，有些 Source 依然需要你在 Spark 属性或者指定的文件中明确配置，就像下面这样。

```
master.source.jvm.class=org.apache.spark.metrics.source.JvmSource
```

5. 获取实例的属性

getInstance 方法（见代码清单 5-80）用于获取指定实例的属性。

代码清单5-80　获取指定实例的属性

```
def getInstance(inst: String): Properties = {
  perInstanceSubProperties.get(inst) match {
    case Some(s) => s
    case None => perInstanceSubProperties.getOrElse(DEFAULT_PREFIX, new
        Properties)
  }
}
```

根据代码清单 5-80，getInstance 方法优先从 perInstanceSubProperties 中获取指定实例的属性，如果 perInstanceSubProperties 中没有存储对应的实例，则返回默认实例（即 *）的属性。

5.8.2　MetricsSystem 中的常用方法

MetricsSystem 中的有些方法使用频度较高，所以本节对这些方法进行介绍。

1. 构建度量源的注册名

buildRegistryName 方法（见代码清单 5-81）用于给 Source 生成向 MetricRegistry 中注册的注册名。

代码清单5-81　构建度量源的注册名

```
private[spark] def buildRegistryName(source: Source): String = {
  val metricsNamespace = conf.get(METRICS_NAMESPACE).orElse(conf.getOption
    ("spark.app.id"))
  val executorId = conf.getOption("spark.executor.id")
  val defaultName = MetricRegistry.name(source.sourceName)
  if (instance == "driver" || instance == "executor") {
    if (metricsNamespace.isDefined && executorId.isDefined) {
      MetricRegistry.name(metricsNamespace.get, executorId.get, source.
        sourceName)
    } else {
      if (metricsNamespace.isEmpty) {
        logWarning(s"Using default name $defaultName for source because neither " +
          s"${METRICS_NAMESPACE.key} nor spark.app.id is set.")
      }
      if (executorId.isEmpty) {
        logWarning(s"Using default name $defaultName for source because spark.
          executor.id is " +
          s"not set.")
      }
      defaultName
    }
  } else { defaultName }
}
```

根据代码清单 5-81，buildRegistryName 方法涉及的变量有以下几个。

1）metricsNamespace：度量命名空间。可以通过 spark.metrics.namespace 属性进行配置，默认读取 spark.app.id 属性的值。例如，笔者本地以 local 模式执行时，TaskScheduler 会设置这个 spark.app.id 属性为 local-1490149388666。

2）executorId：当前 Executor 的身份标识，通过读取 spark.executor.id 属性获得。例如，笔者本地以 local 模式执行时，spark.executor.id 属性为 driver。

3）defaultName：调用 MetricRegistry 的 name 方法（请参阅附录 D）生成的默认注册名。

有了对以上变量的解释，现在来看看 buildRegistryName 方法的执行逻辑。构建注册名的步骤如下。

① 如果定义了 metricsNamespace 和 executorId，那么调用 MetricRegistry 的 name 方法生成 ${metricsNamespace}.${ executorId }.${ defaultName } 格式的注册名。例如，笔者本地以 local 模式执行时，针对 Source 的实现类 CodegenMetrics，生成的注册名为 local-1490149388666.driver.CodeGenerator。

② 如果 metricsNamespace 或 executorId 没有定义，那么采用 defaultName 为注册名。

2. 注册度量源

registerSource 方法（见代码清单 5-82）用于向 MetricsSystem 中注册度量源。MetricRegistry 的 register 方法的实现请参阅附录 D。

代码清单5-82 注册度量源

```
def registerSource(source: Source) {
  sources += source
  try {
    val regName = buildRegistryName(source)
    registry.register(regName, source.metricRegistry)
  } catch {
    case e: IllegalArgumentException => logInfo("Metrics already registered", e)
  }
}
```

3. 获取 ServletContextHandler

为了将度量系统和 Spark UI 结合起来使用，即将 Spark UI 的 Web 展现也作为度量系统的 Sink 之一，我们需要一个 ServletContextHandler 能在 Spark UI 中接收请求，并且还需要能将度量输出到 Spark UI 的页面。根据 3.4.2 节对 Sink 继承体系的介绍，MetricsServlet 是特质 Sink 的实现之一，但是它还不是一个 ServletContextHandler，因此需有一个转换，正如代码清单 5-83 所实现的那样。

代码清单5-83 将MetricsServlet转换为ServletContextHandler

```
def getServletHandlers: Array[ServletContextHandler] = {
  require(running, "Can only call getServletHandlers on a running MetricsSystem")
  metricsServlet.map(_.getHandlers(conf)).getOrElse(Array())
}
```

代码清单 5-83 调用了 MetricsServlet 的 getHandlers 方法来实现转换，getHandlers 的实现如代码清单 5-84 所示。

代码清单5-84 MetricsServlet的getHandlers方法

```
def getHandlers(conf: SparkConf): Array[ServletContextHandler] = {
  Array[ServletContextHandler](
    createServletHandler(servletPath,
      new ServletParams(request => getMetricsSnapshot(request), "text/json"),
        securityMgr, conf)
  )
}
```

根据代码清单 5-84，getHandlers 也调用了 JettyUtils 的 createServletHandler 方法创建 ServletContextHandler。JettyUtils 的 createServletHandler 方法的实现请参阅附录 C。

5.8.3　启动 MetricsSystem

SparkEnv 会调用 MetricsSystem 的 start 方法启动 MetricsSystem，start 方法的实现如代码清单 5-85 所示。

代码清单5-85　启动MetricsSystem

```
def start() {
  require(!running, "Attempting to start a MetricsSystem that is already running")
  running = true
  StaticSources.allSources.foreach(registerSource)
  registerSources()
  registerSinks()
  sinks.foreach(_.start)
}
```

根据代码清单 5-85，启动 MetricsSystem 的步骤如下。

1）将当前 MetricsSystem 的 running 字段置为 true，即表示 MetricsSystem 已经处于运行状态。

2）将静态的度量来源 CodegenMetrics 和 HiveCatalogMetrics 注册到 MetricRegistry。registerSource 方法的实现如代码清单 5-82 所示。allSources 是一个序列，其定义如下。

```
private[spark] object StaticSources {
val allSources = Seq(CodegenMetrics, HiveCatalogMetrics)
}
```

3）从初始化完成的 MetricsConfig 中获取当前实例的度量来源属性，并调用 registerSources 方法（见代码清单 5-86），将这些度量来源注册到 MetricRegistry。

代码清单5-86　注册度量配置中的度量源

```
private def registerSources() {
  val instConfig = metricsConfig.getInstance(instance)
   val sourceConfigs = metricsConfig.subProperties(instConfig, MetricsSystem.
SOURCE_REGEX)
  // Register all the sources related to instance
  sourceConfigs.foreach { kv =>
    val classPath = kv._2.getProperty("class")
    try {
      val source = Utils.classForName(classPath).newInstance()
      registerSource(source.asInstanceOf[Source])
    } catch {
      case e: Exception => logError("Source class " + classPath + " cannot be
          instantiated", e)
    }
  }
}
```

根据代码清单 5-86，注册度量配置中度量源的步骤如下。

①获取当前实例的度量属性。根据 MetricsConfig 的 getInstance 方法（见代码清单

5-80）的逻辑，当实例不存在时将返回默认实例的属性。以 driver 实例来讲，默认不存在 driver 的实例属性，因此返回 * 对应的属性，即：

```
*->{
  sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
  sink.servlet.path=/metrics/json
}
```

② 匹配正则表达式 ^source\.(.+)\.(.+)，获取所有度量源更细粒度的实例及属性。

③ 使用实例的 class 属性，通过 Java 反射生成度量源的实例，并调用 registerSource 方法将此度量源注册到 MetricRegistry。

④ 从初始化完成的 MetricsConfig 中获取当前实例的度量输出属性，并将这些度量输出注册到 sinks。registerSinks 方法的实现如代码清单 5-87 所示。

代码清单5-87　注册度量配置中的度量输出

```
private def registerSinks() {
  val instConfig = metricsConfig.getInstance(instance)
  val sinkConfigs = metricsConfig.subProperties(instConfig, MetricsSystem.SINK_REGEX)
  sinkConfigs.foreach { kv =>
    val classPath = kv._2.getProperty("class")
    if (null != classPath) {
      try {
        val sink = Utils.classForName(classPath)
          .getConstructor(classOf[Properties], classOf[MetricRegistry], classOf
              [SecurityManager])
          .newInstance(kv._2, registry, securityMgr)
        if (kv._1 == "servlet") {
          metricsServlet = Some(sink.asInstanceOf[MetricsServlet])
        } else {
          sinks += sink.asInstanceOf[Sink]
        }
      } catch {
        case e: Exception =>
          logError("Sink class " + classPath + " cannot be instantiated")
          throw e
      }
    }
  }
}
```

根据代码清单 5-87，注册度量配置中度量输出的步骤如下。

① 获取当前实例的度量属性。根据 MetricsConfig 的 getInstance 方法（见代码清单 5-80）的逻辑，当实例不存在时将返回默认实例的属性。以 driver 实例来讲，默认不存在 driver 的实例属性，因此返回 * 对应的属性，即：

```
*->{
  sink.servlet.class=org.apache.spark.metrics.sink.MetricsServlet,
  sink.servlet.path=/metrics/json
}
```

② 匹配正则表达式 ^sink\.(.+)\.(.+)，获取所有度量输出更细粒度的实例及属性。根据 subProperties 方法的逻辑，最后得到：

```
Map(servlet -> {class=org.apache.spark.metrics.sink.MetricsServlet, path=/metrics/json})
```

③ 使用实例的 class 属性，通过 Java 反射生成度量输出的实例。如果当前实例是 servlet，则由 metricsServlet 持有此 servlet 的引用，否则将度量输出实例注册到数组缓冲 sinks 中。

④ 启动 sinks 中的全部度量输出实例。

5.9　输出提交协调器

当 Spark 应用程序使用了 Spark SQL（包括 Hive）或者需要将任务的输出保存到 HDFS 时，就会用到输出提交协调器 OutputCommitCoordinator，OutputCommitCoordinator 将决定任务是否可以提交输出到 HDFS。无论是 Driver 还是 Executor，在 SparkEnv 中都包含了子组件 OutputCommitCoordinator。在 Driver 上注册了 OutputCommitCoordinatorEndpoint，所有 Executor 上的 OutputCommitCoordinator 都是通过 OutputCommitCoordinatorEndpoint 的 RpcEndpointRef 来询问 Driver 上的 OutputCommitCoordinator，是否能够将输出提交到 HDFS。

SparkEnv 中创建 OutputCommitCoordinator 的实现如代码清单 5-88 所示。

代码清单5-88　创建OutputCommitCoordinator

```
val outputCommitCoordinator = mockOutputCommitCoordinator.getOrElse {
  new OutputCommitCoordinator(conf, isDriver)
}
val outputCommitCoordinatorRef = registerOrLookupEndpoint("OutputCommitCoordinator",
  new OutputCommitCoordinatorEndpoint(rpcEnv, outputCommitCoordinator))
outputCommitCoordinator.coordinatorRef = Some(outputCommitCoordinatorRef)
```

根据代码清单 5-88，OutputCommitCoordinator 的创建步骤如下。

1）新建 OutputCommitCoordinator 实例。

2）如果当前实例是 Driver，则创建 OutputCommitCoordinatorEndpoint，并且注册到 Dispatcher 中，注册名为 OutputCommitCoordinator。如果当前应用程序是 Executor，则从远端 Driver 实例的 NettyRpcEnv 的 Dispatcher 中查找 OutputCommitCoordinatorEndpoint 的引用。

3）无论是 Driver 还是 Executor，最后都由 OutputCommitCoordinator 的属性 coordinatorRef 持有 OutputCommitCoordinatorEndpoint 的引用。

5.9.1　OutputCommitCoordinatorEndpoint 的实现

OutputCommitCoordinatorEndpoint 的实现非常简单，如代码清单 5-89 所示。

代码清单5-89 OutputCommitCoordinatorEndpoint的实现

```
private[spark] class OutputCommitCoordinatorEndpoint(
    override val rpcEnv: RpcEnv, outputCommitCoordinator: OutputCommitCoordinator)
  extends RpcEndpoint with Logging {
  override def receive: PartialFunction[Any, Unit] = {
    case StopCoordinator =>
      logInfo("OutputCommitCoordinator stopped!")
      stop()
  }
  override def receiveAndReply(context: RpcCallContext): PartialFunction[Any,
      Unit] = {
    case AskPermissionToCommitOutput(stage, partition, attemptNumber) =>
      context.reply(
        outputCommitCoordinator.handleAskPermissionToCommit(stage, partition,
            attemptNumber))
  }
}
```

根据代码清单 5-89，OutputCommitCoordinatorEndpoint 将接收两个消息。

- StopCoordinator：此消息将停止 OutputCommitCoordinatorEndpoint。
- AskPermissionToCommitOutput：此消息将通过 OutputCommitCoordinator 的 handleAskPermissionToCommit 方法（见代码清单 5-90）处理，进而确认客户端是否有权限将输出提交到 HDFS。

5.9.2 OutputCommitCoordinator 的实现

OutputCommitCoordinator 用于判定给定 Stage 的分区任务是否有权限将输出提交到 HDFS，并对同一分区任务的多次任务尝试（TaskAttempt）进行协调。OutputCommitCoordinator 中有以下属性。

- conf：即 SparkConf。
- isDriver：当前节点是否是 Driver。
- coordinatorRef：即 5.9.1 节介绍的 OutputCommitCoordinatorEndpoint 的 NettyRpcEndpointRef 引用。
- NO_AUTHORIZED_COMMITTER：值为 -1 的常量。
- authorizedCommittersByStage：缓存 Stage 的各个分区的任务尝试。类型为 scala.collection.mutable.Map[StageId, Array[TaskAttemptNumber]]。

了解了 OutputCommitCoordinator 的属性，来看看 OutputCommitCoordinator 实现的方法。

1. handleAskPermissionToCommit

handleAskPermissionToCommit 方法（见代码清单 5-90）用于判断给定的任务尝试是否有权限将给定 Stage 的指定分区的数据提交到 HDFS。

代码清单5-90　handleAskPermissionToCommit的实现

```
private[scheduler] def handleAskPermissionToCommit(
    stage: StageId,
    partition: PartitionId,
    attemptNumber: TaskAttemptNumber): Boolean = synchronized {
  authorizedCommittersByStage.get(stage) match {
    case Some(authorizedCommitters) =>
      authorizedCommitters(partition) match {
        case NO_AUTHORIZED_COMMITTER =>
          logDebug(s"Authorizing attemptNumber=$attemptNumber to commit for
              stage=$stage, " +
            s"partition=$partition")
          authorizedCommitters(partition) = attemptNumber
          true
        case existingCommitter =>
          logDebug(s"Denying attemptNumber=$attemptNumber to commit for stage=
              $stage, " +
            s"partition=$partition; existingCommitter = $existingCommitter")
          false
      }
    case None =>
        logDebug(s"Stage $stage has completed, so not allowing attempt number
$attemptNumber of" +
        s"partition $partition to commit")
      false
  }
}
```

根据代码清单 5-90，handleAskPermissionToCommit 的处理步骤如下。

1）从 authorizedCommittersByStage 缓存中找到给定 Stage 的指定分区的 TaskAttemptNumber（实际类型为 Int）。

2）如果第 1）步获取的 TaskAttemptNumber 等于 NO_AUTHORIZED_COMMITTER，则说明当前是首次提交给定 Stage 的指定分区的输出，因此按照第一提交者胜利（first committer wins）策略，给定 TaskAttemptNumber（即 attemptNumber）有权限将给定 Stage 的指定分区的输出提交到 HDFS。为了告诉后来的任务尝试“已经有人捷足先登”，还需要将给定分区的索引与 attemptNumber 的关系保存到 TaskAttemptNumber 数组中。

3）如果第 1）步获取的 TaskAttemptNumber 不等于 NO_AUTHORIZED_COMMITTER，则说明之前已经有任务尝试将给定 Stage 的指定分区的输出提交到 HDFS，因此按照第一提交者胜利（first committer wins）策略，给定 TaskAttemptNumber（即 attemptNumber）没有权限将给定 Stage 的指定分区的输出提交到 HDFS。

2. isEmpty

isEmpty 方法（见代码清单 5-91）用于判断 authorizedCommittersByStage 是否为空。

代码清单5-91　判断authorizedCommittersByStage是否为空

```
def isEmpty: Boolean = {
```

```
    authorizedCommittersByStage.isEmpty
  }
```

3. canCommit

canCommit 方法（见代码清单 5-92）用于向 OutputCommitCoordinatorEndpoint 发送 AskPermissionToCommitOutput，并根据 OutputCommitCoordinatorEndpoint 的响应确认是否有权限将 Stage 的指定分区的输出提交到 HDFS 上。

代码清单5-92 canCommit的实现

```
def canCommit(
    stage: StageId,
    partition: PartitionId,
    attemptNumber: TaskAttemptNumber): Boolean = {
  val msg = AskPermissionToCommitOutput(stage, partition, attemptNumber)
  coordinatorRef match {
    case Some(endpointRef) =>
      endpointRef.askWithRetry[Boolean](msg)  //询问是否有权限将Stage的指定分区的输出提
          交到HDFS上
    case None =>
      logError(
        "canCommit called after coordinator was stopped (is SparkEnv shutdown in
            progress)?")
      false
  }
}
```

4. stageStart

stageStart 方法（见代码清单 5-93）用于启动给定 Stage 的输出提交到 HDFS 的协调机制，其实质为创建给定 Stage 的对应 TaskAttemptNumber 数组，并将 TaskAttemptNumber 数组中的所有 TaskAttemptNumber 置为 NO_AUTHORIZED_COMMITTER。

代码清单5-93 stageStart的实现

```
private[scheduler] def stageStart(
    stage: StageId,
    maxPartitionId: Int): Unit = {
  val arr = new Array[TaskAttemptNumber](maxPartitionId + 1)
  java.util.Arrays.fill(arr, NO_AUTHORIZED_COMMITTER)
  synchronized {
    authorizedCommittersByStage(stage) = arr
  }
}
```

5. stageEnd

stageEnd 方法（见代码清单 5-94）用于停止给定 Stage 的输出提交到 HDFS 的协调机制，其实质为将给定 Stage 及对应的 TaskAttemptNumber 数组从 authorizedCommitters-ByStage 中删除。

代码清单5-94　stageEnd的实现

```
private[scheduler] def stageEnd(stage: StageId): Unit = synchronized {
  authorizedCommittersByStage.remove(stage)
}
```

6. taskCompleted

给定 Stage 的指定分区的任务执行完成后将调用 taskCompleted 方法（见代码清单 5-95）。

代码清单5-95　taskCompleted的实现

```
private[scheduler] def taskCompleted(
    stage: StageId,
    partition: PartitionId,
    attemptNumber: TaskAttemptNumber,
    reason: TaskEndReason): Unit = synchronized {
  val authorizedCommitters = authorizedCommittersByStage.getOrElse(stage, {
    logDebug(s"Ignoring task completion for completed stage")
    return
  })
  reason match {
    case Success =>
    // The task output has been committed successfully
    case denied: TaskCommitDenied =>
      logInfo(s"Task was denied committing, stage: $stage, partition: $partition, " +
        s"attempt: $attemptNumber")
    case otherReason =>
      if (authorizedCommitters(partition) == attemptNumber) {
        logDebug(s"Authorized committer (attemptNumber=$attemptNumber,
            stage=$stage, " +
          s"partition=$partition) failed; clearing lock")
        authorizedCommitters(partition) = NO_AUTHORIZED_COMMITTER
      }
  }
}
```

根据代码清单 5-95，有三种情况可视为任务完成。

- **任务执行成功**：此时 reason 等于特质 TaskEndReason 的子类 Success。
- **任务提交被拒绝**：此时 reason 等于特质 TaskEndReason 的子类 TaskCommitDenied。
- **其他原因**：此时 reason 等于特质 TaskEndReason 的其他子类。针对这种原因，需要将给定 Stage 的对应 TaskAttemptNumber 数组中指定分区的值修改为 NO_AUTHORIZED_COMMITTER，以便于之后的任务尝试能够有权限提交。

7. stop

stop 方法（见代码清单 5-96）通过向 OutputCommitCoordinatorEndpoint 发送 StopCoordinator 消息以停止 OutputCommitCoordinatorEndpoint，然后清空 authorizedCommitters-ByStage。

代码清单5-96　停止OutputCommitCoordinator

```
def stop(): Unit = synchronized {
```

```
    if (isDriver) {
      coordinatorRef.foreach(_ send StopCoordinator)
      coordinatorRef = None
      authorizedCommittersByStage.clear()
    }
  }
```

5.9.3 OutputCommitCoordinator 的工作原理

经过对 OutputCommitCoordinatorEndpoint 和 OutputCommitCoordinator 的详细介绍，OutputCommitCoordinator 决定任务是否可以提交输出到 HDFS 的工作原理可以用图 5-15 来总结说明。

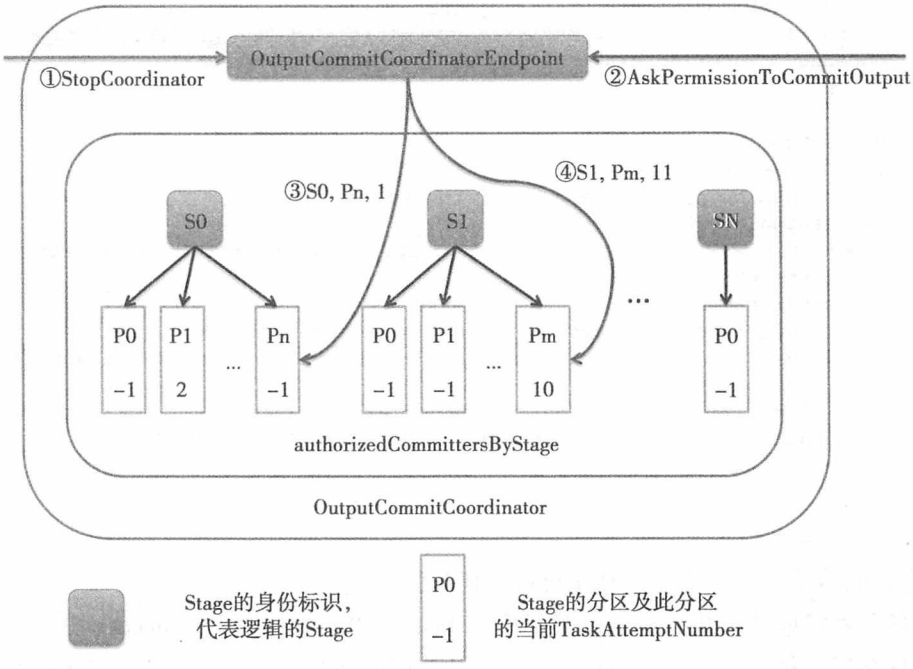


图 5-15 OutputCommitCoordinator 的工作原理

图 5-15 展现了 OutputCommitCoordinator 的逻辑结构和工作原理。authorizedCommittersByStage 缓存了每一个 Stage 及其分区的内存结构，S0、S1 及 SN 代表不同的 Stage，P0、P1 则代表 Stage 的各个分区，Pn、Pm 说明每个 Stage 内的分区数量不同。图中每个序号的含义如下。

记号①：表示 OutputCommitCoordinatorEndpoint 收到 StopCoordinator 消息，OutputCommitCoordinatorEndpoint 将调用父类 RpcEndpoint 的 stop 方法（见代码清单 5-2），RpcEndpoint 的 stop 方法实际又调用了 NettyRpcEnv 的 stop 方法（见代码清单 5-45），停止 OutputCommitCoordinatorEndpoint 的工作。

记号②：表示 OutputCommitCoordinatorEndpoint 在接收到 AskPermissionToCommitOutput 消息后，调用 OutputCommitCoordinator 的 handleAskPermissionToCommit 方法判断给定的任务尝试是否有权限将给定 Stage 的指定分区的数据提交到 HDFS。

记号③：表示 AskPermissionToCommitOutput 消息携带的 Stage 为 S0，分区为 Pn，任务尝试号为 1，handleAskPermissionToCommit 方法执行时发现阶段 S0 的分区 Pn 还未有任务尝试占用（即值为 -1），则允许当前任务尝试将阶段 S0 的分区 Pn 的数据提交到 HDFS 并且将 Pn 的值设置为 1。

记号④：表示 AskPermissionToCommitOutput 消息携带的 Stage 为 S1，分区为 Pm，任务尝试号为 11，handleAskPermissionToCommit 方法执行时发现阶段 S1 的分区 Pm 已被其他任务尝试占用（占用此分区的任务尝试号为 10），则不允许当前任务尝试将阶段 S1 的分区 Pm 的数据提交到 HDFS。

5.10　创建 SparkEnv

当 SparkEnv 内的所有组件都实例化完毕，将正式构造 SparkEnv。除了本章介绍的各种组件外，SparkEnv 内部还有以下几个成员属性。

- isStopped：当前 SparkEnv 是否停止的状态。
- pythonWorkers：所有 python 实现的 Worker 的缓存。
- hadoopJobMetadata：HadoopRDD 进行任务切分时所需要的元数据的软引用。例如，HadoopFileRDD 将使用 hadoopJobMetadata 缓存 JobConf 和 InputFormat。
- driverTmpDir：如果当前 SparkEnv 处于 Driver 实例中，那么将创建 Driver 的临时目录。创建 Driver 的临时目录的代码如下。

```
if (isDriver) {
  val sparkFilesDir = Utils.createTempDir(Utils.getLocalDir(conf), "userFiles").
    getAbsolutePath
  envInstance.driverTmpDir = Some(sparkFilesDir)
}
```

上面代码中创建临时目录是由工具类 Utils 的 createTempDir 方法实现的，其具体实现请参阅附录 A。

5.11　小结

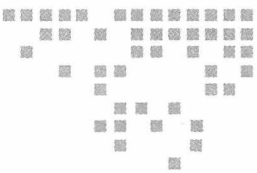
本章首先对构成 SparkEnv 的各个组件进行了展示，然后逐一讲解每个组件的创建。对有些组件的工作原理进行了详细的分析，这是因为这些组件的功能可以供其他组件使用，如安全管理器、RPC 环境、广播管理器、度量系统等。有些组件的功能跟任务的计算非常紧密，所以把这些组件的详细原理放在计算引擎的相关章节中，例如，shuffle 管理器、内

存管理器等。有些组件的功能都属于存储体系，这些组件的具体内容也统一放入存储体系的相关章节中，例如，块管理器、BlockManagerMaster、块传输服务等。

通过对 RPC 环境的讲解，读者将了解到服务端和客户端实现的细节，很多组件的通信都离不开 RPC 环境，例如，块管理器、map 任务输出跟踪器、输出提交协调器等。

广播管理器对广播对象的广播实际是通过对存储体系的读写完成的。广播的真正含义在于写入存储体系的备份数量，这个问题将在存储体系中详细介绍。很多组件中的对象都需要广播管理器进行广播，例如，Hadoop 配置、序列化后的 RDD、Job 及 ShuffleDependency 等。

度量系统可以将多种度量来源的度量数据输出到多种度量输出。还通过给 MetricsServlet 创建 ServletContextHandler，将度量系统与 Spark UI 结合起来使用。



第6章 Chapter 6

存储体系

“天行健，君子以自强不息；地势坤，君子以厚德载物。”

——《易经》

本章导读

我们生活的地球上，存在着种类繁多的生物。这些生物的生存离不开阳光、空气和水。中国传统文化认为，天空风云多变，变幻无穷，非常强健，是君子向往的；无论是花鸟鱼虫，还是山川河海，都由大地承载，亦是君子追求的。古人常说：“天高任鸟飞，海阔凭鱼跃。”但是鸟飞得久了也得回巢，鱼跃得再高还是得回到水中。世间万物，无论你是什么，依然需要大地的承载。大地是安静的，也是非常包容的。

Spark的存储体系在整个Spark架构中与大地在生态圈中的作用非常类似。Spark的存储体系从Spark集群的横向角度看，贯穿了集群中的每个实例。从单个节点来看，Spark的存储体系隶属于SparkEnv。由于存储体系的内容非常丰富，很有必要开辟单独的一章内容进行详细分析。笔者将所有涉及存储的内容都集中在本章，以便于对存储体系有好的抽象，也便于读者对其内容有更宏观的认识。

早在5.5节介绍BroadcastManager的时候，我们就知道对广播对象进行广播离不开存储体系的支持。除此之外，无论是SparkContext的初始化，还是任务的提交与执行，始终都离不开存储体系。Spark为了避免Hadoop频繁读写磁盘造成磁盘I/O成为性能瓶颈，优先会将配置信息、计算结果等数据存入内存，这极大地提升了系统的执行效率。正是因为这一关键决策，才让Spark能在大数据应用中表现出优秀的计算能力。

本章主要讲解的内容如下。

- 存储体系的基本概念。
- 存储体系架构。
- BlockManager 中的各个组件。

6.1 存储体系概述

本书在 5.7 节曾介绍过存储体系的创建，那时只为帮助读者了解 SparkEnv，现在是时候对 Spark 的存储体系进行详细的分析了。简单来讲，Spark 存储体系是各个 Driver 和 Executor 实例中的 BlockManager 所组成的。但是从一个整体出发，把各个节点的 BlockManager 看成存储体系的一部分，那么存储体系还有更多衍生的内容，比如块传输服务、map 任务输出跟踪器、Shuffle 管理器等。

6.1.1 存储体系架构

在正式介绍存储体系的内容之前，如果能对存储体系从整体上有个宏观的认识，无疑有利于读者对后续内容的理解。图 6-1 能够从整体上表示存储体系架构。

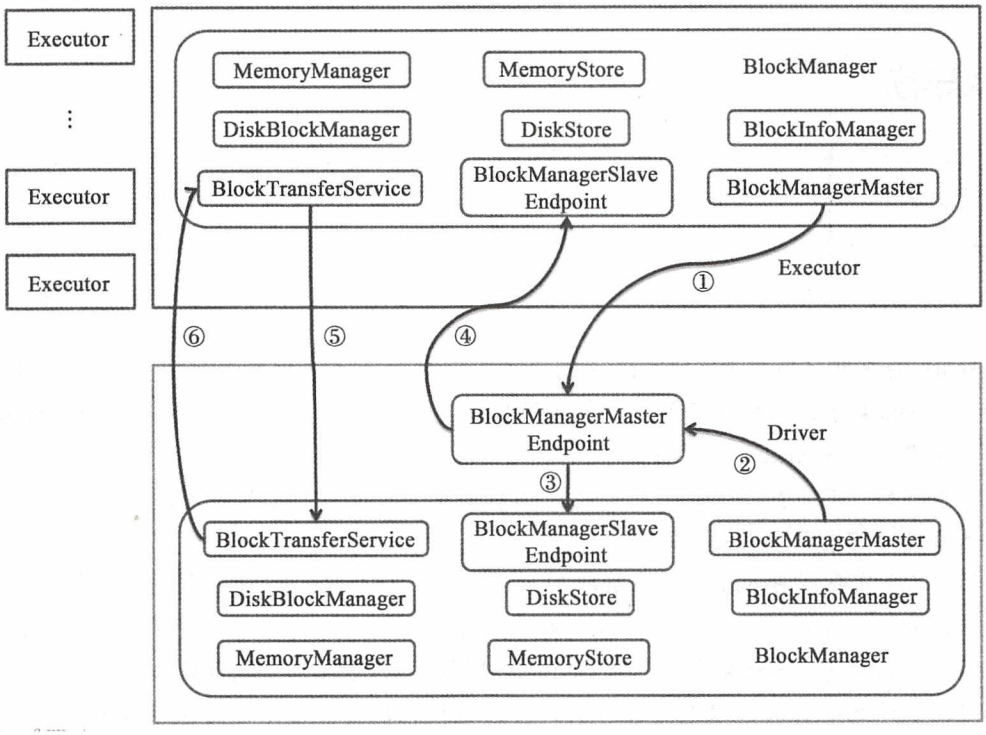


图 6-1 存储体系架构

从图 6-1 我们可以看到，BlockManager 依托于很多组件的服务，这些组件包括以下几项。

1）BlockManagerMaster：代理 BlockManager 与 Driver 上的 BlockManagerMasterEndpoint 通信。图 6-1 中的记号①表示 Executor 节点上的 BlockManager 通过 BlockManagerMaster 与 BlockManagerMasterEndpoint 进行通信，记号②表示 Driver 节点上的 BlockManager 通过 BlockManagerMaster 与 BlockManagerMasterEndpoint 进行通信。这些通信的内容有很多，例如，注册 BlockManager、更新 Block 信息、获取 Block 的位置（即 Block 所在的 BlockManager）、删除 Executor 等。BlockManagerMaster 之所以能够和 BlockManagerMasterEndpoint 通信，是因为它持有了 BlockManagerMasterEndpoint 的 RpcEndpointRef。

2）BlockManagerMasterEndpoint：由 Driver 上的 SparkEnv 负责创建和注册到 Driver 的 RpcEnv 中。BlockManagerMasterEndpoint 只存在于 Driver 的 SparkEnv 中，Driver 或 Executor 上的 BlockManagerMaster 的 driverEndpoint 属性将持有 BlockManagerMasterEndpoint 的 RpcEndpointRef。BlockManagerMasterEndpoint 主要对各个节点上的 BlockManager、BlockManager 与 Executor 的映射关系及 Block 位置信息（即 Block 所在的 BlockManager）等进行管理。

3）BlockManagerSlaveEndpoint：每个 Executor 或 Driver 的 SparkEnv 中都有属于自己的 BlockManagerSlaveEndpoint，分别由各自的 SparkEnv 负责创建和注册到各自的 RpcEnv 中。Driver 或 Executor 都存在各自的 BlockManagerSlaveEndpoint，并由各自 BlockManager 的 slaveEndpoint 属性持有各自 BlockManagerSlaveEndpoint 的 RpcEndpointRef。BlockManagerSlaveEndpoint 将接收 BlockManagerMasterEndpoint 下发的命令。图 6-1 中的记号③表示 BlockManagerMasterEndpoint 向 Driver 节点上的 BlockManagerSlaveEndpoint 下发命令，记号④表示 BlockManagerMasterEndpoint 向 Executor 节点上的 BlockManagerSlaveEndpoint 下发命令。这些下发命令有很多，例如，删除 Block、获取 Block 状态、获取匹配的 BlockId 等。

4）SerializerManager：序列化管理器。SerializerManager 的内容已在 5.4 节详细分析过，此处不再赘述。

5）MemoryManager：内存管理器。负责对单个节点上内存的分配与回收。

6）MapOutputTracker：map 任务输出跟踪器。其实现与工作原理已在 5.6 节详细介绍过。

7）ShuffleManager：Shuffle 管理器。由于 Shuffle 与计算的关系更为紧密，所以留在第 8 章详细介绍。

8）BlockTransferService：块传输服务。此组件也与 Shuffle 相关联，主要用于不同阶段的任务之间的 Block 数据的传输与读写。例如，map 任务所在节点的 BlockTransferService 给 Shuffle 对应的 reduce 任务提供下载 map 中间输出结果的服务。

9）shuffleClient：Shuffle 的客户端。与 BlockTransferService 配合使用。图 6-1 中的记号⑤表示 Executor 上的 shuffleClient 通过 Driver 上的 BlockTransferService 提供的服务上传和下载 Block，记号⑥表示 Driver 上的 shuffleClient 通过 Executor 上的 BlockTransferService 提供的服务上传和下载 Block。此外，不同 Executor 节点上的 BlockTransfer-Service 和 shuffleClient 之间也可以互相上传、下载 Block。

10）SecurityManager：安全管理器。具体实现已在 5.2 节介绍。

11）DiskBlockManager：磁盘块管理器。对磁盘上的文件及目录的读写操作进行管理。

12）BlockInfoManager：块信息管理器。负责对 Block 的元数据及锁资源进行管理。

13）MemoryStore：内存存储。依赖于 MemoryManager，负责对 Block 的内存存储。

14）DiskStore：磁盘存储。依赖于 DiskBlockManager，负责对 Block 的磁盘存储。

小贴士：在 Spark 1.x.x 版本中还提供了 TachyonStore，TachyonStore 负责向 Tachyon 中存储数据。Tachyon 也诞生于 UC Berkeley 的 AMP 实验室，是以内存为中心的高容错的分布式文件系统，能够为内存计算框架（比如 Spark、MapReduce 等）提供可靠的内存级的文件共享服务。从软件栈的层次来看，Tachyon 是位于现有大数据计算框架和大数据存储系统之间的独立的一层。它利用底层文件系统作为备份，对于上层应用来说，Tachyon 就是一个分布式文件系统。自 Spark 2.0.0 版本开始，TachyonStore 已经被移除。Tachyon 发展到 1.0.0 版本时改名为 Alluxio。更多 Alluxio 的信息，请访问 http://www.alluxio.org。

6.1.2 基本概念

存储体系中有一些概念，需要详细说明。正所谓“磨刀不误砍柴工”，有了对这些概念的理解，后面的内容学习起来会更加容易。

1. BlockManager 的唯一标识 BlockManagerId

根据之前的了解，我们知道在 Driver 或者 Executor 中有任务执行的环境 SparkEnv。每个 SparkEnv 中都有 BlockManager，这些 BlockManager 位于不同的节点和实例上。BlockManager 之间需要通过 RpcEnv、shuffleClient 及 BlockTransferService 相互通信，所以大家需要互相认识，正如每个人都有身份证号一样，每个 BlockManager 都有其在 Spark 集群内的唯一标识。BlockManagerId 就是 BlockManager 的身份证。

Spark 通过 BlockManagerId 中的 host、port、executorId 等信息来区分 BlockManager。BlockManagerId 中的属性包括以下几项。

- host_：主机域名或 IP。
- port_：此端口实际使用了 BlockManager 中的 BlockTransferService 对外服务的端口。
- executorId_：当前 BlockManager 所在的实例的 ID。如果实例是 Driver，那么 ID 为 driver，否则由 Master 负责给各个 Executor 分配，ID 格式为 app- 日期格式字符串 - 数字。

- topologyInfo_：拓扑信息。

理解了 BlockManagerId 中的属性，下面来看看它提供的方法。

- executorId：返回 executorId_ 的值。
- hostPort：返回 host:port 格式的字符串。
- host：返回 host_ 的值。
- port：返回 port_ 的值。
- topologyInfo：返回 topologyInfo_ 的值。
- isDriver：当前 BlockManager 所在的实例是否是 Driver。此方法实际根据 executorId_ 的值是否是 driver 来判断。
- writeExternal：将 BlockManagerId 的所有信息序列化后写到外部二进制流中。writeExternal 的实现如代码清单 6-1 所示。

代码清单6-1 writeExternal的实现

```
override def writeExternal(out: ObjectOutput): Unit = Utils.tryOrIOException {
  out.writeUTF(executorId_)
  out.writeUTF(host_)
  out.writeInt(port_)
  out.writeBoolean(topologyInfo_.isDefined)
  // we only write topologyInfo if we have it
  topologyInfo.foreach(out.writeUTF(_))
}
```

- readExternal：从外部二进制流中读取 BlockManagerId 的所有信息。readExternal 的实现如代码清单 6-2 所示。

代码清单6-2 readExternal的实现

```
override def readExternal(in: ObjectInput): Unit = Utils.tryOrIOException {
  executorId_ = in.readUTF()
  host_ = in.readUTF()
  port_ = in.readInt()
  val isTopologyInfoAvailable = in.readBoolean()
  topologyInfo_ = if (isTopologyInfoAvailable) Option(in.readUTF()) else None
}
```

2. 块的唯一标识 BlockId

了解操作系统存储的读者应该知道，文件系统的文件在存储到磁盘上时，都是以块为单位写入的。操作系统的块是以固定的大小读写的，例如，512 字节、1024 字节、2048 字节、4096 字节等。

在 Spark 的存储体系中，数据的读写也是以块为单位，只不过这个块并非操作系统的块，而是设计用于 Spark 存储体系的块。每个 Block 都有唯一的标识，Spark 把这个标识抽象为 BlockId。抽象类 BlockId 的定义如代码清单 6-3 所示。

代码清单6-3 BlockId的定义

```
@DeveloperApi
sealed abstract class BlockId {
  def name: String
  def asRDDId: Option[RDDBlockId] = if (isRDD) Some(asInstanceOf[RDDBlockId])
    else None
  def isRDD: Boolean = isInstanceOf[RDDBlockId]
  def isShuffle: Boolean = isInstanceOf[ShuffleBlockId]
  def isBroadcast: Boolean = isInstanceOf[BroadcastBlockId]
  override def toString: String = name
  override def hashCode: Int = name.hashCode
  override def equals(other: Any): Boolean = other match {
    case o: BlockId => getClass == o.getClass && name.equals(o.name)
    case _ => false
  }
}
```

根据代码清单 6-3，BlockId 中定义了以下方法。

- name：Block 全局唯一的标识名。
- isRDD：当前 BlockId 是否是 RDDBlockId。
- asRDDId：将当前 BlockId 转换为 RDDBlockId。如果当前 BlockId 是 RDDBlockId，则转换为 RDDBlockId，否则返回 None。
- isShuffle：当前 BlockId 是否是 ShuffleBlockId。
- isBroadcast：当前 BlockId 是否是 BroadcastBlockId。

BlockId 有很多子类，例如，RDDBlockId、ShuffleBlockId、BroadcastBlockId 等。BroadcastBlockId 曾在 5.5 节介绍广播管理器 BroadcastManager 时介绍过，BlockId 的子类都是用相似的方式实现的。

3. 存储级别 StorageLevel

前文提到，Spark 的存储体系包括磁盘存储与内存存储。Spark 将内存又分为堆外内存和堆内存。有些数据块本身支持序列化及反序列化，有些数据块还支持备份与复制。Spark 存储体系将以上这些数据块的不同特性抽象为存储级别（StorageLevel）。

分析 StorageLevel，也是从其成员属性开始。StorageLevel 中的成员属性如下。

- _useDisk：能否写入磁盘。当 Block 的 StorageLevel 中的 _useDisk 为 true 时，存储体系将允许 Block 写入磁盘。
- _useMemory：能否写入堆内存。当 Block 的 StorageLevel 中的 _useMemory 为 true 时，存储体系将允许 Block 写入堆内存。
- _useOffHeap：能否写入堆外内存。当 Block 的 StorageLevel 中的 _useOffHeap 为 true 时，存储体系将允许 Block 写入堆外内存。
- _deserialized：是否需要对 Block 反序列化。当 Block 本身经过了序列化后，Block 的 StorageLevel 中的 _deserialized 将被设置为 true，即可以对 Block 进行反序列化。

- _replication：Block 的复制份数。Block 的 StorageLevel 中的 _replication 默认等于 1，可以在构造 Block 的 StorageLevel 时明确指定 _replication 的数量。当 _replication 大于 1 时，Block 除了在本地的存储体系中写入一份，还会复制到其他不同节点的存储体系中写入，达到复制备份的目的。

有了对 StorageLevel 中属性的了解，现在来看看 StorageLevel 提供了哪些方法。

- useDisk：能否写入磁盘。实际直接返回了 _useDisk 的值。
- useMemory：能否写入堆内存。实际直接返回了 _useMemory 的值。
- useOffHeap：能否写入堆外内存。实际直接返回了 _useOffHeap 的值。
- deserialized：是否需要对 Block 反序列化。实际直接返回了 _deserialized 的值。
- replication：复制份数。实际直接返回了 _replication 的值。
- memoryMode：内存模式。如果 useOffHeap 为 true，则返回枚举值 MemoryMode.OFF_HEAP，否则返回枚举值 MemoryMode.ON_HEAP。
- clone：对当前 StorageLevel 进行克隆，并返回克隆的 StorageLevel。
- isValid：当前的 StorageLevel 是否有效。判断的条件如下。

```
(useMemory || useDisk) && (replication > 0)
```

- toInt：将当前 StorageLevel 转换为整型表示。toInt 的实现如代码清单 6-4 所示。

代码清单6-4 toInt的实现

```
def toInt: Int = {
  var ret = 0
  if (_useDisk) {
    ret |= 8
  }
  if (_useMemory) {
    ret |= 4
  }
  if (_useOffHeap) {
    ret |= 2
  }
  if (_deserialized) {
    ret |= 1
  }
  ret
}
```

根据代码清单 6-4，toInt 方法实际是把 StorageLevel 的 _useDisk、_useMemory、_useOffHeap、_deserialized 这四个属性设置到四位数字的各个状态位。例如，1000 表示存储级别为允许写入磁盘；1100 表示存储级别为允许写入磁盘和堆内存；1111 表示存储级别为允许写入磁盘、堆内存及堆外内存，并且需要反序列化。

- writeExternal：将 StorageLevel 首先通过 toInt 方法将 _useDisk、_useMemory、_useOffHeap、_deserialized 四个属性设置到四位数的状态位，然后与 _replication 一起被序列化写入外部二进制流。writeExternal 的实现如代码清单 6-5 所示。

书籍是人类进步的阶梯

代码清单6-5 writeExternal的实现

```
override def writeExternal(out: ObjectOutput): Unit = Utils.tryOrIOException {
  out.writeByte(toInt)
  out.writeByte(_replication)
}
```

- readExternal：从外部二进制流中读取 StorageLevel 的各个属性。readExternal 的实现如代码清单 6-6 所示。

代码清单6-6 readExternal的实现

```
override def readExternal(in: ObjectInput): Unit = Utils.tryOrIOException {
  val flags = in.readByte()
  _useDisk = (flags & 8) != 0
  _useMemory = (flags & 4) != 0
  _useOffHeap = (flags & 2) != 0
  _deserialized = (flags & 1) != 0
  _replication = in.readByte()
}
```

了解了 StorageLevel 原生类提供的这些方法，我们再来看看 StorageLevel 的伴生对象。由于 StorageLevel 的构造器是私有的，所以 StorageLevel 的伴生对象中已经预先定义了很多存储体系需要的 StorageLevel，如代码清单 6-7 所示。

代码清单6-7 内置的StorageLevel

```
val NONE = new StorageLevel(false, false, false, false)
val DISK_ONLY = new StorageLevel(true, false, false, false)
val DISK_ONLY_2 = new StorageLevel(true, false, false, false, 2)
val MEMORY_ONLY = new StorageLevel(false, true, false, true)
val MEMORY_ONLY_2 = new StorageLevel(false, true, false, true, 2)
val MEMORY_ONLY_SER = new StorageLevel(false, true, false, false)
val MEMORY_ONLY_SER_2 = new StorageLevel(false, true, false, false, 2)
val MEMORY_AND_DISK = new StorageLevel(true, true, false, true)
val MEMORY_AND_DISK_2 = new StorageLevel(true, true, false, true, 2)
val MEMORY_AND_DISK_SER = new StorageLevel(true, true, false, false)
val MEMORY_AND_DISK_SER_2 = new StorageLevel(true, true, false, false, 2)
val OFF_HEAP = new StorageLevel(true, true, true, false, 1)
```

代码清单 6-7 所示的代码调用 StorageLevel 的构造器创建了多种存储级别。StorageLevel 私有构造器的参数从左至右分别为 _useDisk、_useMemory、_useOffHeap、_deserialized、_replication。

4. 块信息 BlockInfo

BlockInfo 用于描述块的元数据信息，包括存储级别、Block 类型、大小、锁信息等。按照惯例，我们先看看 BlockInfo 的成员属性。

- level：BlockInfo 所描述的 Block 的存储级别，即 StorageLevel。
- classTag：BlockInfo 所描述的 Block 的类型。

书籍是人类进步的阶梯

- tellMaster：BlockInfo 所描述的 Block 是否需要告知 Master。
- _size：BlockInfo 所描述的 Block 的大小。
- _readerCount：BlockInfo 所描述的 Block 被锁定读取的次数。
- _writerTask：任务尝试在对 Block 进行写操作前，首先必须获得对应 BlockInfo 的写锁。_writerTask 用于保存任务尝试的 ID（每个任务在实际执行时，会多次尝试，每次尝试都会分配一个 ID）。

有了对 BlockInfo 的了解，现在来看看 BlockInfo 提供的方法。

- size 与 size_：对 _size 的读、写。
- readerCount 与 readerCount_：对 _readerCount 的读、写。
- writerTask 与 writerTask_：对 _writerTask 的读、写。

5. BlockResult

BlockResult 用于封装从本地的 BlockManager 中获取的 Block 数据及与 Block 相关联的度量数据。BlockResult 中有以下属性。

- data：Block 及与 Block 相关联的度量数据。
- readMethod：读取 Block 的方法。readMethod 采用枚举类型 DataReadMethod 提供的 Memory、Disk、Hadoop、Network 四个枚举值。
- bytes：读取的 Block 的字节长度。

6. BlockStatus

样例类 BlockStatus 用于封装 Block 的状态信息，包括以下属性。

- storageLevel：即 Block 的 StorageLevel。
- memSize：Block 占用的内存大小。
- diskSize：Block 占用的磁盘大小。
- isCached：是否存储到存储体系中，即 memSize 与 diskSize 的大小之和是否大于 0。

6.2 Block 信息管理器

按照 BlockInfoManager 的命名，笔者将本节的标题定为“Block 信息管理器”，BlockInfoManager 的确对 BlockInfo 进行了一些简单的管理，但是 BlockInfoManager 将主要对 Block 的锁资源进行管理，当读者阅读了本节的内容，将会同意笔者的观点。

6.2.1 Block 锁的基本概念

BlockInfoManager 是 BlockManager 内部的子组件之一，BlockInfoManager 对 Block 的锁管理采用了共享锁与排他锁，其中读锁是共享锁，写锁是排他锁。表 6-1 展示了 Block 的读锁与写锁之间的互斥性。

书籍是人类进步的阶梯

表 6-1 Block 的读锁与写锁之间的互斥性

	读 锁	写 锁
读锁	S	X
写锁	X	X

表 6-1 中采用了介绍数据库锁的常规表示方法：S 代表共享，X 代表排他。

要了解 BlockInfoManager 的工作原理，首先应从了解它的属性开始。BlockInfoManager 的成员属性包括以下几个。

- infos：BlockId 与 BlockInfo 之间映射关系的缓存。
- writeLocksByTask：每次任务执行尝试的标识 TaskAttemptId 与执行获取的 Block 的写锁之间的映射关系。TaskAttemptId 与写锁之间是一对多的关系，即一次任务尝试执行会获取零到多个 Block 的写锁。类型 TaskAttemptId 的本质是 Long 类型，其定义如下。

```
private type TaskAttemptId = Long
```

- readLocksByTask：每次任务尝试执行的标识 TaskAttemptId 与执行获取的 Block 的读锁之间的映射关系。TaskAttemptId 与读锁之间是一对多的关系，即一次任务尝试执行会获取零到多个 Block 的读锁，并且会记录对于同一个 Block 的读锁的占用次数。

有了对 BlockInfoManager 中属性的了解，我们可以通过图 6-2 来展示 BlockInfoManager 对 Block 的锁管理。

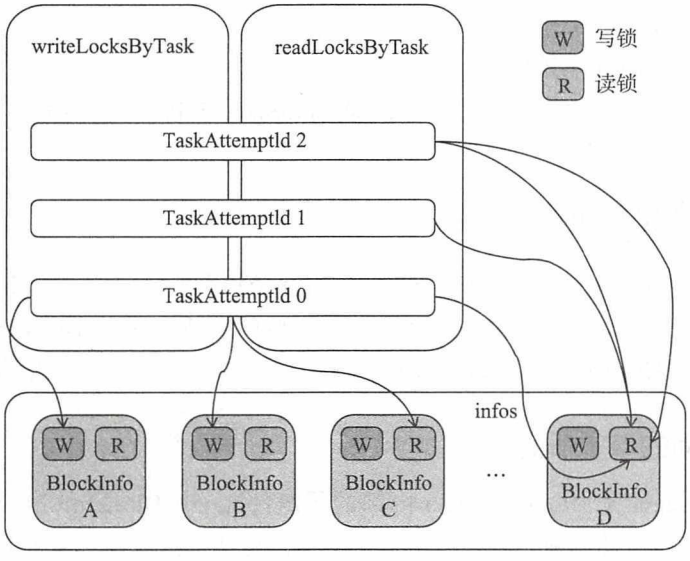


图 6-2 BlockInfoManager 对 Block 的锁管理

书籍是人类进步的阶梯

从图 6-2，我们能直观地知道以下内容。

- 由 TaskAttemptId 0 标记的任务尝试执行线程获取了 BlockInfo A 和 BlockInfo B 的写锁，并且获取了 BlockInfo C 和 BlockInfo D 的读锁。
- 由 TaskAttemptId 1 标记的任务尝试执行线程获取了 BlockInfo D 的读锁。
- 由 TaskAttemptId 2 标记的任务尝试执行线程多次获取了 BlockInfo D 的读锁，这说明 Block 的读锁是可以重入的。

根据图中三个任务尝试执行线程获取锁的不同展示，我们可以知道，一个任务尝试执行线程可以同时获得零到多个不同 Block 的写锁或零到多个不同 Block 的读锁，但不能同时获得同一个 Block 的读锁与写锁（这点和一些数据库的设计是不同的）。读锁是可以重入的，但是写锁不能重入。

6.2.2 Block 锁的实现

有了 BlockInfoManager 对 Block 的锁管理的了解，现在来看看 BlockInfoManager 提供的方法是如何实现 Block 的锁管理机制的。

1）registerTask：注册 TaskAttemptId。registerTask 的实现如代码清单 6-8 所示。

代码清单6-8 注册TaskAttemptId

```
def registerTask(taskAttemptId: TaskAttemptId): Unit = synchronized {
  require(!readLocksByTask.contains(taskAttemptId),
    s"Task attempt $taskAttemptId is already registered")
  readLocksByTask(taskAttemptId) = ConcurrentHashMultiset.create()
}
```

2）currentTaskAttemptId：获取任务上下文 TaskContext 中当前正在执行的任务尝试的 TaskAttemptId。如果任务上下文 TaskContext 中没有任务尝试的 TaskAttemptId，那么返回 BlockInfo.NON_TASK_WRITER。currentTaskAttemptId 方法的实现如代码清单 6-9 所示。

代码清单6-9 currentTaskAttemptId的实现

```
private def currentTaskAttemptId: TaskAttemptId = {
  Option(TaskContext.get()).map(_.taskAttemptId()).getOrElse(BlockInfo.NON_TASK_
    WRITER)
}
```

3）lockForReading：锁定读。lockForReading 的实现如代码清单 6-10 所示。

代码清单6-10 lockForReading的实现

```
def lockForReading(
    blockId: BlockId,
    blocking: Boolean = true): Option[BlockInfo] = synchronized {
  logTrace(s"Task $currentTaskAttemptId trying to acquire read lock for $blockId")
  do {
    infos.get(blockId) match { // 从infos中获取BlockId对应的BlockInfo
      case None => return None
```

书籍是人类进步的阶梯

```
        case Some(info) =>
          if (info.writerTask == BlockInfo.NO_WRITER) { // 由当前任务尝试线程持有读锁并
              返回BlockInfo
            info.readerCount += 1
            readLocksByTask(currentTaskAttemptId).add(blockId)
            logTrace(s"Task $currentTaskAttemptId acquired read lock for $blockId")
            return Some(info)
          }
      }
      if (blocking) {
        wait() // 等待占用写锁的任务尝试线程释放Block的写锁后唤醒当前线程
      }
    } while (blocking)
    None
  }
```

根据代码清单 6-10，lockForReading 方法的执行步骤如下。

① 从 infos 中获取 BlockId 对应的 BlockInfo。如果缓存 infos 中没有对应的 BlockInfo，则返回 None，否则进入下一步。

② 如果 Block 的写锁没有被其他任务尝试线程占用，则由当前任务尝试线程持有读锁并返回 BlockInfo，否则进入下一步。

③ 如果允许阻塞（即 blocking 为 true），那么当前线程将等待，直到占用写锁的线程释放 Block 的写锁后唤醒当前线程。如果占有写锁的线程一直不释放写锁，那么当前线程将出现“饥饿”状况，即可能无限期等待下去。

4）lockForWriting：锁定写。lockForWriting 的实现如代码清单 6-11 所示。

代码清单6-11 lockForWriting的实现

```
def lockForWriting(
    blockId: BlockId,
    blocking: Boolean = true): Option[BlockInfo] = synchronized {
  logTrace(s"Task $currentTaskAttemptId trying to acquire write lock for
      $blockId")
  do {
    infos.get(blockId) match { // 从infos中获取BlockId对应的BlockInfo
      case None => return None
      case Some(info) =>
        if (info.writerTask == BlockInfo.NO_WRITER && info.readerCount == 0) {
          info.writerTask = currentTaskAttemptId // 由当前任务尝试线程持有写锁
          writeLocksByTask.addBinding(currentTaskAttemptId, blockId)
          logTrace(s"Task $currentTaskAttemptId acquired write lock for $blockId")
          return Some(info)
        }
    }
    if (blocking) {
      wait() // 等待占用写锁的线程释放Block的写锁后唤醒当前线程
    }
  } while (blocking)
  None
}
```

根据代码清单 6-11，lockForWriting 方法的执行步骤如下。

① 从 infos 中获取 BlockId 对应的 BlockInfo。如果缓存 infos 中没有对应的 BlockInfo，则返回 None，否则进入下一步。

② 如果 Block 的写锁没有被其他任务尝试线程占用，且没有线程正在读取此 Block，则由当前任务尝试线程持有写锁并返回 BlockInfo，否则进入下一步。写锁没有被占用并且没有线程正在读取此 Block 的条件也说明了任务尝试执行线程不能同时获得同一个 Block 的读锁与写锁。

③ 如果允许阻塞（即 blocking 为 true），那么当前线程将等待，直到占用写锁的线程释放 Block 的写锁后唤醒当前线程。如果占有写锁的线程一直不释放写锁，那么当前线程将出现“饥饿”状况，即可能无限期等待下去。

小贴士：lockForReading 和 lockForWriting 这两个方法共同实现了写锁与写锁、写锁与读锁之间的互斥性，同时也实现了读锁与读锁之间的共享。此外，这两个方法都提供了阻塞的方式。这种方式在读锁或写锁的争用较少或锁的持有时间都非常短暂，能够带来一定的性能提升。如果获取锁的线程发现锁被占用，就立即失败，然而这个锁很快又被释放了，结果是获取锁的线程无法正常执行。如果获取锁的线程可以等待的话，很快它就发现自己能重新获得锁了，然后推进当前线程继续执行。

- get：获取 BlockId 对应的 BlockInfo。get 方法的实现如代码清单 6-12 所示。

代码清单6-12 获取BlockId对应的BlockInfo

```
private[storage] def get(blockId: BlockId): Option[BlockInfo] = synchronized {
  infos.get(blockId)
}
```

- unlock：释放 BlockId 对应的 Block 上的锁。

代码清单6-13 释放BlockId对应的Block上的锁

```
def unlock(blockId: BlockId): Unit = synchronized {
  logTrace(s"Task $currentTaskAttemptId releasing lock for $blockId")
  val info = get(blockId).getOrElse { // 获取BlockId对应的BlockInfo
    throw new IllegalStateException(s"Block $blockId not found")
  }
  if (info.writerTask != BlockInfo.NO_WRITER) {
    info.writerTask = BlockInfo.NO_WRITER // 释放当前Block的写锁
    writeLocksByTask.removeBinding(currentTaskAttemptId, blockId)
  } else { // 释放当前Block的读锁
    assert(info.readerCount > 0, s"Block $blockId is not locked for reading")
    info.readerCount -= 1
    val countsForTask = readLocksByTask(currentTaskAttemptId)
    val newPinCountForTask: Int = countsForTask.remove(blockId, 1) - 1
    assert(newPinCountForTask >= 0,
      s"Task $currentTaskAttemptId release lock on block $blockId more times than
          it acquired it")
```

```
    }
    notifyAll()
  }
```

根据代码清单 6-13，unlock 方法的执行步骤如下。

① 获取 BlockId 对应的 BlockInfo。

② 如果当前任务尝试线程已经获得了 Block 的写锁，则释放当前 Block 的写锁。

③ 如果当前任务尝试线程没有获得 Block 的写锁，则释放当前 Block 的读锁。释放读锁实际是减少当前任务尝试线程已经获取的 Block 的读锁次数。

小贴士：上述代码中的 newPinCountForTask 是当前尝试执行线程占有 BlockId 对应 Block 的读锁的次数与 1 的差值，如果大于等于 0，则说明当前尝试执行线程多次获得了这个读锁。

- downgradeLock：锁降级。downgradeLock 的实现如代码清单 6-14 所示。

代码清单6-14　downgradeLock的实现

```
def downgradeLock(blockId: BlockId): Unit = synchronized {
  logTrace(s"Task $currentTaskAttemptId downgrading write lock for $blockId")
  val info = get(blockId).get // 获取BlockId对应的BlockInfo
  require(info.writerTask == currentTaskAttemptId,
    s"Task $currentTaskAttemptId tried to downgrade a write lock that it does not hold on" +
      s" block $blockId")
  unlock(blockId) // 释放Block的写锁
  val lockOutcome = lockForReading(blockId, blocking = false) // 获取Block的读锁
  assert(lockOutcome.isDefined)
}
```

根据代码清单 6-14，锁降级的过程如下。

① 获取 BlockId 对应的 BlockInfo。

② 调用 unlock 方法释放当前任务尝试线程从 BlockId 对应 Block 获取的写锁。

③ 由于已经释放了 BlockId 对应 Block 的写锁，所以用非阻塞方式获取 BlockId 对应 Block 的读锁。

7）lockNewBlockForWriting：写新 Block 时获得写锁。lockNewBlockForWriting 的实现如代码清单 6-15 所示。

代码清单6-15　写新Block时获得写锁

```
def lockNewBlockForWriting(
    blockId: BlockId,
    newBlockInfo: BlockInfo): Boolean = synchronized {
  logTrace(s"Task $currentTaskAttemptId trying to put $blockId")
  lockForReading(blockId) match { // 获取Block的读锁
    case Some(info) =>
      false
    case None =>
```

```
        infos(blockId) = newBlockInfo
        lockForWriting(blockId) // 获取Block的写锁
        true
      }
    }
```

根据代码清单 6-15，lockNewBlockForWriting 的执行步骤如下。

① 获取 BlockId 对应的 Block 的读锁。

② 如果上一步能够获取到 Block 的读锁，则说明 BlockId 对应的 Block 已经存在。这种情况发生在多个线程在写同一个 Block 时产生竞争，已经有线程率先一步，当前线程将没有必要再获得写锁，只需要返回 false。

③ 如果第①步没有获取到 Block 的读锁，则说明 BlockId 对应的 Block 还不存在。这种情况下，当前线程首先将 BlockId 与新的 BlockInfo 的映射关系放入 infos，然后获取 BlockId 对应的 Block 的写锁，最后返回 true。

8）releaseAllLocksForTask：释放给定的任务尝试线程所占用的所有 Block 的锁，并通知所有等待获取锁的线程。由于此方法不常使用且实现简单，留给感兴趣的读者自行查阅。

9）size：返回 infos 的大小，即所有 Block 的数量。

10）entries：以迭代器形式返回 infos。entries 的实现如代码清单 6-16 所示。

代码清单6-16 以迭代器形式返回infos

```
def entries: Iterator[(BlockId, BlockInfo)] = synchronized {
  infos.toArray.toIterator
}
```

11）removeBlock：移除 BlockId 对应的 BlockInfo。removeBlock 方法的实现如代码清单 6-17 所示。

代码清单6-17 移除BlockId对应的BlockInfo

```
def removeBlock(blockId: BlockId): Unit = synchronized {
  logTrace(s"Task $currentTaskAttemptId trying to remove block $blockId")
  infos.get(blockId) match { // 获取BlockId对应的BlockInfo
    case Some(blockInfo) =>
      if (blockInfo.writerTask != currentTaskAttemptId) {
        throw new IllegalStateException(
          s"Task $currentTaskAttemptId called remove() on block $blockId without
              a write lock")
      } else { // 移除BlockInfo
        infos.remove(blockId)
        blockInfo.readerCount = 0
        blockInfo.writerTask = BlockInfo.NO_WRITER
        writeLocksByTask.removeBinding(currentTaskAttemptId, blockId)
      }
    case None =>
      throw new IllegalArgumentException(
        s"Task $currentTaskAttemptId called remove() on non-existent block
            $blockId")
```

```
    }
    notifyAll() // 通知所有在BlockId对应的Block的锁上等待的线程
  }
```

根据代码清单 6-17，removeBlock 的执行步骤如下。

① 获取 BlockId 对应的 BlockInfo。

② 如果对 BlockInfo 正在写入的任务尝试线程是当前线程的话，当前线程才有权利去移除 BlockInfo。移除 BlockInfo 的操作如下。

- 将 BlockInfo 从 infos 中移除。
- 将 BlockInfo 的读线程数清零。
- 将 BlockInfo 的 writerTask 置为 BlockInfo.NO_WRITER。
- 将任务尝试线程与 BlockId 的关系清除。

③ 通知所有在 BlockId 对应的 Block 的锁上等待的线程。

12）clear：清除 BlockInfoManager 中的所有信息，并通知所有在 BlockInfoManager 管理的 Block 的锁上等待的线程。

6.3 磁盘 Block 管理器

DiskBlockManager 是存储体系的成员之一，它负责为逻辑的 Block 与数据写入磁盘的位置之间建立逻辑的映射关系。对于不了解的事物，我们应该先看看它定义了哪些属性。DiskBlockManager 的属性如下。

- conf：即 SparkConf。
- deleteFilesOnStop：停止 DiskBlockManager 的时候是否删除本地目录的布尔类型标记。当不指定外部的 ShuffleClient（即 spark.shuffle.service.enabled 属性为 false）或者当前实例是 Driver 时，此属性为 true。
- localDirs：本地目录的数组。

6.3.1 本地目录结构

localDirs 是 DiskBlockManager 管理的本地目录数组。localDirs 是通过调用 createLocal-Dirs 方法（见代码清单 6-18）创建的本地目录数组，其实质是调用了 Utils 工具类的 getConfigured-LocalDirs 方法获取本地路径（Utils 的详细介绍请阅读附录 A。getConfiguredLocalDirs 方法默认获取 spark.local.dir 属性或者系统属性 java.io.tmpdir 指定的目录，目录可能有多个），并在每个路径下创建以 blockmgr- 为前缀，UUID 为后缀的随机字符串的子目录，例如，blockmgr-4949e19c-490c-48fc-ad6a-d80f4dbe73df。

代码清单6-18 创建DiskBlockManager的本地目录

```
private def createLocalDirs(conf: SparkConf): Array[File] = {
```

```
    Utils.getConfiguredLocalDirs(conf).flatMap { rootDir =>
      try {
        val localDir = Utils.createDirectory(rootDir, "blockmgr")
        logInfo(s"Created local directory at $localDir")
        Some(localDir)
      } catch {
        case e: IOException =>
          logError(s"Failed to create local dir in $rootDir. Ignoring this directory.", e)
          None
      }
    }
  }
```

- subDirsPerLocalDir：磁盘存储 DiskStore 的本地子目录的数量。可以通过 spark.diskStore.subDirectories 属性配置，默认为 64。
- subDirs：DiskStore 的本地子目录的二维数组，即 File[localDirs.length][subDirsPerLocalDir]。
- shutdownHook：此属性的作用是在初始化 DiskBlockManager 时，调用 addShutdownHook 方法（见代码清单 6-19），为 DiskBlockManager 设置好关闭钩子。

代码清单6-19 为DiskBlockManager设置关闭钩子

```
private def addShutdownHook(): AnyRef = {
  logDebug("Adding shutdown hook") // force eager creation of logger
  ShutdownHookManager.addShutdownHook(ShutdownHookManager.TEMP_DIR_SHUTDOWN_
      PRIORITY + 1) { () =>
    logInfo("Shutdown hook called")
    DiskBlockManager.this.doStop()
  }
}
```

有了对 localDirs 的了解，让我们通过图 6-3 对 DiskBlockManager 管理的文件目录有一个整体的认识。

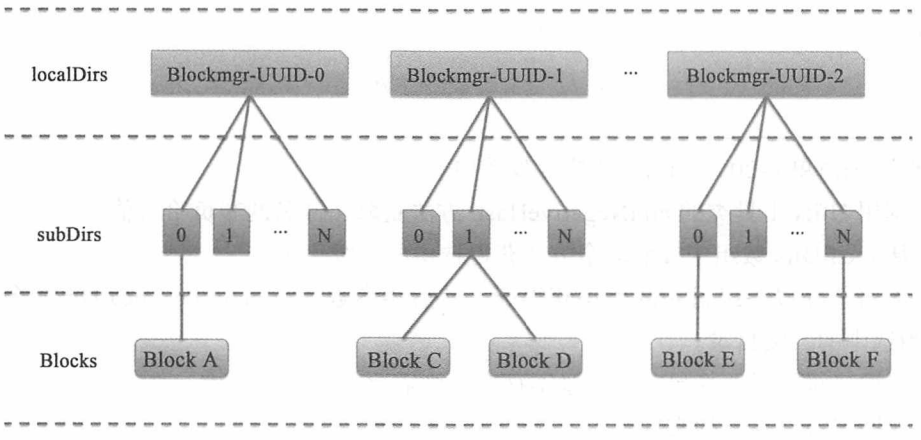


图 6-3 DiskBlockManager 管理的文件目录结构

图 6-3 中显示了一级目录，即 localDirs 数组。图中的一级目录名称都采用了简单的示意表示，例如，Blockmgr-UUID-0、Blockmgr-UUID-1、Blockmgr-UUID-2，代表每个文件夹名称由 Blockmgr- 和 UUID 生成的随机串组成，且此随机串不相同。subDirs 表示 DiskBlockManager 管理的二级目录，每个一级目录下都有 N 个二级目录，这里使用 N 代表 subDirsPerLocalDir 属性的大小。每个二级目录下有若干个 Block 的文件，有些二级目录下可能暂时还没有 Block 文件。有了对 DiskBlockManager 管理的文件目录结构的整体认识，下面来看看 DiskBlockManager 提供了哪些方法来构成和管理这些目录。

6.3.2 DiskBlockManager 提供的方法

1. getFile(filename: String)

此方法根据指定的文件名获取文件。此方法的实现如代码清单 6-20 所示。

代码清单6-20 根据指定的文件名获取文件

```
def getFile(filename: String): File = {
  val hash = Utils.nonNegativeHash(filename) // 获取文件名的非负哈希值
  val dirId = hash % localDirs.length        // 按照取余方式获得选中的一级目录
  // 获得的余数作为选中的二级目录
  val subDirId = (hash / localDirs.length) % subDirsPerLocalDir
  val subDir = subDirs(dirId).synchronized { // 获取二级目录
    val old = subDirs(dirId)(subDirId)
    if (old != null) {
      old
    } else {
      val newDir = new File(localDirs(dirId), "%02x".format(subDirId))
      if (!newDir.exists() && !newDir.mkdir()) {
        throw new IOException(s"Failed to create local dir in $newDir.")
      }
      subDirs(dirId)(subDirId) = newDir
      newDir
    }
  }

  new File(subDir, filename) // 返回二级目录下的文件
}
```

根据代码清单 6-20，getFile 的执行步骤如下。

1）调用 Utils 工具类的 nonNegativeHash 方法获取文件名的非负哈希值。

2）从 localDirs 数组中按照取余方式获得选中的一级目录。

3）哈希值除以一级目录的大小获得商，然后用商数与 subDirsPerLocalDir 取余获得的余数作为选中的二级目录。

4）获取二级目录。如果二级目录不存在，则需要创建二级目录。

5）返回二级目录下的文件。

2. getFile(blockId: BlockId)

此方法根据 BlockId 获取文件。此方法的实现如代码清单 6-21 所示。

代码清单6-21 根据BlockId获取文件

```
def getFile(blockId: BlockId): File = getFile(blockId.name)
```

根据代码清单 6-21，getFile(blockId: BlockId) 实际是以 BlockId 的 name 为参数，通过调用 getFile(filename: String) 方法实现的。

3. containsBlock

此方法用于检查本地 localDirs 目录中是否包含 BlockId 对应的文件。containsBlock 的实现如代码清单 6-22 所示。

代码清单6-22 containsBlock的实现

```
def containsBlock(blockId: BlockId): Boolean = {
  getFile(blockId.name).exists()
}
```

4. getAllFiles

此方法用于获取本地 localDirs 目录中的所有文件。getAllFiles 的实现如代码清单 6-23 所示。

代码清单6-23 getAllFiles的实现

```
def getAllFiles(): Seq[File] = {
  subDirs.flatMap { dir =>
    dir.synchronized {
      dir.clone()
    }
  }.filter(_ != null).flatMap { dir =>
    val files = dir.listFiles()
    if (files != null) files else Seq.empty
  }
}
```

根据代码清单 6-23 可以知道，getAllFiles 为了保证线程安全，采用了同步 + 克隆的方式。

5. getAllBlocks

此方法用于获取本地 localDirs 目录中所有 Block 的 BlockId。getAllBlocks 方法的实现如代码清单 6-24 所示。

代码清单6-24 getAllBlocks的实现

```
def getAllBlocks(): Seq[BlockId] = {
  getAllFiles().map(f => BlockId(f.getName))
}
```

6. createTempLocalBlock

此方法用于为中间结果创建唯一的 BlockId 和文件，此文件将用于保存本地 Block 的数据。createTempLocalBlock 的实现如代码清单 6-25 所示。

代码清单6-25 createTempLocalBlock的实现

```
def createTempLocalBlock(): (TempLocalBlockId, File) = {
  var blockId = new TempLocalBlockId(UUID.randomUUID())
  while (getFile(blockId).exists()) {
    blockId = new TempLocalBlockId(UUID.randomUUID())
  }
  (blockId, getFile(blockId))
}
```

7. createTempShuffleBlock

此方法创建唯一的 BlockId 和文件，用来存储 Shuffle 中间结果（即 map 任务的输出）。createTempShuffleBlock 的实现如代码清单 6-26 所示。

代码清单6-26 createTempShuffleBlock的实现

```
def createTempShuffleBlock(): (TempShuffleBlockId, File) = {
  var blockId = new TempShuffleBlockId(UUID.randomUUID())
  while (getFile(blockId).exists()) {
    blockId = new TempShuffleBlockId(UUID.randomUUID())
  }
  (blockId, getFile(blockId))
}
```

8. stop

此方法用于正常停止 DiskBlockManager。stop 的实现如代码清单 6-27 所示。

代码清单6-27 停止DiskBlockManager

```
private[spark] def stop() {
  try {
    ShutdownHookManager.removeShutdownHook(shutdownHook)
  } catch {
    case e: Exception =>
      logError(s"Exception while removing shutdown hook.", e)
  }
  doStop()
}
```

从代码清单 6-27 看到，实际停止 DiskBlockManager 的方法为 doStop，其实现如代码清单 6-28 所示。

代码清单6-28 doStop的实现

```
private def doStop(): Unit = {
  if (deleteFilesOnStop) {
```

```
    localDirs.foreach { localDir =>
      if (localDir.isDirectory() && localDir.exists()) {
        try {
          if (!ShutdownHookManager.hasRootAsShutdownDeleteDir(localDir)) {
            Utils.deleteRecursively(localDir)
          }
        } catch {
          case e: Exception =>
            logError(s"Exception while deleting local spark dir: $localDir", e)
        }
      }
    }
  }
```

根据代码清单6-28，doStop的主要逻辑是遍历localDirs数组中的一级目录，并调用工具类Utils的deleteRecursively方法（具体实现请参阅附录A），递归删除一级目录及其子目录或子文件。

6.4 磁盘存储DiskStore

DiskStore负责将Block存储到磁盘。由于DiskStore依赖于DiskBlockManager的服务，所以我们在上一节先介绍了DiskBlockManager。在Spark 1.x.x版本中，BlockStore提供了对磁盘存储DiskStore、内存存储MemoryStore及Tachyon（即Alluxio）存储TachyonStore的统一规范，DiskStore、MemoryStore和TachyonStore都是具体的实现。但从Spark 2.0.0版本开始，取消了TachyonStore，而且也不再使用BlockStore提供统一的接口规范，DiskStore和MemoryStore都是分别实现的。本节将介绍DiskStore的实现。

我们首先从了解DiskStore的属性开始，逐步理解DiskStore发挥的作用，以及DiskStore与DiskBlockManager之间的关系。DiskStore的属性有如下几项。

- conf：即SparkConf。
- diskManager：即磁盘Block管理器DiskBlockManager。
- minMemoryMapBytes：读取磁盘中的Block时，是直接读取还是使用FileChannel的内存镜像映射方法读取的阈值。

小贴士：什么是FileChannel的内存镜像映射方法？在Java NIO中，FileChannel的map方法所提供的快速读写技术，其实质上是将通道所连接的数据节点中的全部或部分数据直接映射到内存的一个Buffer中，而这个内存Buffer块就是节点数据的映像，你直接对这个Buffer进行修改，会影响到节点数据。这个Buffer叫做MappedBuffer，即镜像Buffer。由于是内存镜像，因此处理速度快。

有了对DiskStore中属性的了解，现在我们来看看DiskStore提供的方法。

1. getSize

此方法用于获取给定 BlockId 所对应 Block 的大小。getSize 的实现如代码清单 6-29 所示。

代码清单6-29　获取Block文件的大小

```
def getSize(blockId: BlockId): Long = {
  diskManager.getFile(blockId.name).length
}
```

根据代码清单 6-29，可以看到 DiskStore 的 getSize 方法实质是对 DiskBlockManager 的 getFile 方法的调用，在获取了 File 之后，取得 File 的大小。

2. contains

此方法用于判断本地磁盘存储路径下是否包含给定 BlockId 所对应的 Block 文件。contains 的实现如代码清单 6-30 所示。

代码清单6-30　Block文件是否存在

```
def contains(blockId: BlockId): Boolean = {
  val file = diskManager.getFile(blockId.name)
  file.exists()
}
```

3. remove

此方法用于删除给定 BlockId 所对应的 Block 文件。remove 的实现如代码清单 6-31 所示。

代码清单6-31　删除Block文件

```
def remove(blockId: BlockId): Boolean = {
  val file = diskManager.getFile(blockId.name)
  if (file.exists()) {
    val ret = file.delete()
    if (!ret) {
      logWarning(s"Error deleting ${file.getPath()}")
    }
    ret
  } else {
    false
  }
}
```

4. put

此方法用于将 BlockId 所对应的 Block 写入磁盘。put 的实现如代码清单 6-32 所示。

代码清单6-32　写入Block文件

```
def put(blockId: BlockId)(writeFunc: FileOutputStream => Unit): Unit = {
```

```
    if (contains(blockId)) { // 判断给定BlockId所对应的Block文件是否存在
      throw new IllegalStateException(s"Block $blockId is already present in the
          disk store")
    }
    logDebug(s"Attempting to put block $blockId")
    val startTime = System.currentTimeMillis
    val file = diskManager.getFile(blockId) // 获取BlockId所对应的Block文件
    val fileOutputStream = new FileOutputStream(file)
    var threwException: Boolean = true
    try {
      writeFunc(fileOutputStream) // 对Block文件写入
      threwException = false
    } finally {
      try {
        Closeables.close(fileOutputStream, threwException)
      } finally {
         if (threwException) {
          remove(blockId)
        }
      }
    }
    val finishTime = System.currentTimeMillis
    logDebug("Block %s stored as %s file on disk in %d ms".format(
      file.getName,
      Utils.bytesToString(file.length()),
      finishTime - startTime))
  }
```

根据代码清单 6-32，put 的执行步骤如下。

1）调用 contains 方法判断给定 BlockId 所对应的 Block 文件是否存在，当存在时进入下一步。

2）调用 DiskBlockManager 的 getFile 方法获取 BlockId 所对应的 Block 文件，并打开文件输出流。

3）调用回调函数 writeFunc，对 Block 文件写入。当写入失败时，还需要调用 remove 方法删除 BlockId 所对应的 Block 文件。

5. putBytes

此方法用于将 BlockId 所对应的 Block 写入磁盘，Block 的内容已经封装为 ChunkedByteBuffer。putBytes 的实现如代码清单 6-33 所示。

代码清单6-33 putBytes的实现

```
def putBytes(blockId: BlockId, bytes: ChunkedByteBuffer): Unit = {
  put(blockId) { fileOutputStream =>
    val channel = fileOutputStream.getChannel
    Utils.tryWithSafeFinally {
      bytes.writeFully(channel)
    } {
```

```
        channel.close()
      }
    }
  }
```

根据代码清单 6-33，在 putBytes 方法中设置了回调函数，并调用 put 方法。根据 put 方法的实现，我们知道最后会回调外部的回调函数。此回调函数中使用了工具类 Utils 的 tryWithSafeFinally 方法，其详细介绍请阅读附录 A。

6. getBytes

此方法用于读取给定 BlockId 所对应的 Block，并封装为 ChunkedByteBuffer 返回。getBytes 的实现如代码清单 6-34 所示。

代码清单6-34　读取给定BlockId所对应的Block

```
def getBytes(blockId: BlockId): ChunkedByteBuffer = {
  val file = diskManager.getFile(blockId.name)
  val channel = new RandomAccessFile(file, "r").getChannel
  Utils.tryWithSafeFinally {
    // For small files, directly read rather than memory map
    if (file.length < minMemoryMapBytes) {
      val buf = ByteBuffer.allocate(file.length.toInt)
      channel.position(0)
      while (buf.remaining() != 0) {
        if (channel.read(buf) == -1) {
          throw new IOException("Reached EOF before filling buffer\n" +
            s"offset=0\nfile=${file.getAbsolutePath}\nbuf.remaining=${buf.remaining}")
        }
      }
      buf.flip()
      new ChunkedByteBuffer(buf)
    } else {
      new ChunkedByteBuffer(channel.map(MapMode.READ_ONLY, 0, file.length))
    }
  } {
    channel.close()
  }
}
```

6.3 节和 6.4 节介绍的都是磁盘存储的内容，下面将介绍内存存储。

6.5 内存管理器

Spark 与 Hadoop 的重要区别之一就在于对内存的使用。Hadoop 只将内存作为计算资源，Spark 除将内存作为计算资源外，还将内存的一部分纳入到存储体系中。Spark 使用 MemoryManager 对存储体系和内存计算所使用的内存进行管理。由于本章主讲存储体系的内容，因此本节将只介绍和存储体系相关的部分，涉及计算引擎的内容，将放在第 8 章介绍。

6.5.1 内存池模型

内存池好比游泳馆的游泳池，只不过游泳池装的是水，内存池装的是内存。游泳馆往往不止一个游泳池，Spark 的存储体系的每个存储节点上也不止一个内存池。内存池实质上是对物理内存的逻辑规划，协助 Spark 任务在运行时合理地使用内存资源。Spark 将内存从逻辑上区分为堆内存和堆外内存，称为内存模式（MemoryMode）。枚举类型 MemoryMode 中定义了堆内存和堆外内存，代码如下。

```
@Private
public enum MemoryMode {
  ON_HEAP,
  OFF_HEAP
}
```

这里的堆内存并不能与 JVM 中的 Java 堆直接画等号，它只是 JVM 堆内存的一部分。堆外内存则是 Spark 使用 sun.misc.Unsafe 的 API 直接在工作节点的系统内存中开辟的空间。无论是哪种内存，都需要一个内存池对内存进行资源管理，抽象类 MemoryPool（见代码清单 6-35）定义了内存池的规范。

代码清单6-35 内存池的规范

```
private[memory] abstract class MemoryPool(lock: Object) {
  @GuardedBy("lock")
  private[this] var _poolSize: Long = 0
  final def poolSize: Long = lock.synchronized {
    _poolSize
  }
  final def memoryFree: Long = lock.synchronized {
    _poolSize - memoryUsed
  }
  final def incrementPoolSize(delta: Long): Unit = lock.synchronized {
    require(delta >= 0)
    _poolSize += delta
  }
  final def decrementPoolSize(delta: Long): Unit = lock.synchronized {
    require(delta >= 0)
    require(delta <= _poolSize)
    require(_poolSize - delta >= memoryUsed)
    _poolSize -= delta
  }
  def memoryUsed: Long
}
```

根据代码清单 6-35，内存池的基本属性如下。

- lock：对内存池提供线程安全保证的锁对象。
- _poolSize：内存池的大小（单位为字节）。

内存池的基本方法如下。

- poolSize：返回内存池的大小（即 _poolSize，单位为字节）的方法。

- memoryUsed：获取已经使用的内存大小（单位为字节）。此方法需要 MemoryPool 的子类实现。
- memoryFree：获取内存池的空闲空间（即 _poolSize 减去 memoryUsed 的大小，单位为字节）。
- incrementPoolSize：给内存池扩展 delta 给定的大小（单位为字节）。delta 必须为正整数。
- decrementPoolSize：将内存池缩小 delta 给定的大小（单位为字节）。delta 必须为正整数且 _poolSize 与 delta 的差要大于等于 memoryUsed（即已经使用的内存不能从内存池中移除）。

根据 MemoryPool 所定义的内存池规范，我们可以用图 6-4 来表示 MemoryPool 的内存模型。

图 6-4 所示的内存模型完全是逻辑上的划分，并不是物理上的内存划分。

Spark 中一共有两种 MemoryPool 的具体实现，其继承体系如图 6-5 所示。

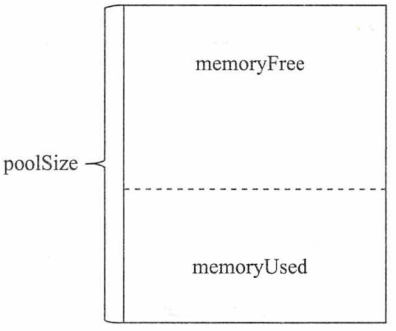


图 6-4　MemoryPool 的内存模型

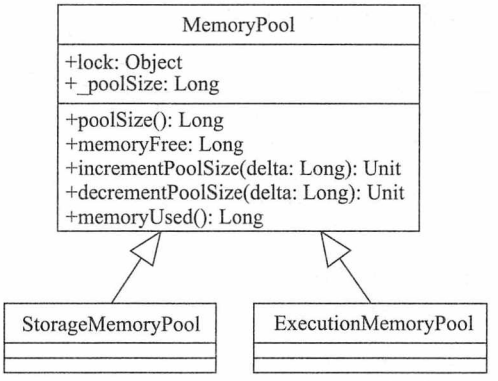


图 6-5　MemoryPool 的继承体系

图 6-5 中的 StorageMemoryPool 是存储体系用到的内存池，而 ExecutionMemoryPool 则是计算引擎用到的内存池。

6.5.2 StorageMemoryPool 详解

Spark 既将内存作为存储体系的一部分，又作为计算引擎所需要的计算资源，因此 MemoryPool 既有用于存储体系的实现类 StorageMemoryPool，又有用于计算的 Execution MemoryPool。由于本章内容着重介绍存储体系，所以本节只介绍 StorageMemoryPool，而 ExecutionMemoryPool 则留在第 8 章介绍。

StorageMemoryPool 是对用于存储的物理内存的逻辑抽象，通过对存储内存的逻辑管理，提高 Spark 存储体系对内存的使用效率。要理解 StorageMemoryPool，先从其属性开始。StorageMemoryPool 继承了 MemoryPool 的 lock 和 _poolSize 两个属性，还增加了一些

特有的属性。

- memoryMode：内存模式（MemoryMode）。由此可以断定用于存储的内存池包括堆内存的内存池和堆外内存的内存池。StorageMemoryPool 的堆外内存和堆内存都是逻辑上的区分，前者是使用 sun.misc.Unsafe 的 API 分配的系统内存，后者是系统分配给 JVM 堆内存的一部分。
- poolName：内存池的名称。如果 memoryMode 是 MemoryMode.ON_HEAP，则内存池名称为 on-heap storage。如果 memoryMode 是 MemoryMode.OFF_HEAP，则内存池名称为 off-heap storage。
- _memoryUsed：已经使用的内存大小（单位为字节）。
- _memoryStore：当前 StorageMemoryPool 所关联的 MemoryStore。MemoryStore 将在 6.6 节详细介绍。

了解了 StorageMemoryPool 的属性，现在来看看 StorageMemoryPool 中的方法。

- memoryUsed：此方法实现了由 MemoryPool 定义的 memoryUsed 接口，实际返回了 _memoryUsed 属性的值。
- memoryStore：获取当前 StorageMemoryPool 所关联的 MemoryStore，实际返回了 _memoryStore 属性引用的 MemoryStore。
- setMemoryStore：设置当前 StorageMemoryPool 所关联的 MemoryStore，实际设置了 _memoryStore 属性。
- releaseAllMemory：释放当前内存池的所有内存，即将 _memoryUsed 设置为 0。

以上方法的实现都很简单，下面介绍一些需要详细分析的方法。

1. acquireMemory

acquireMemory 方法（见代码清单 6-36）用于给 BlockId 对应的 Block 获取 numBytes 指定大小的内存。

代码清单6-36 获取存储Block所需的内存

```
def acquireMemory(blockId: BlockId, numBytes: Long): Boolean = lock.synchronized {
  val numBytesToFree = math.max(0, numBytes - memoryFree)
  acquireMemory(blockId, numBytes, numBytesToFree)
}
```

根据代码清单 6-36，此方法首先计算要申请的内存大小 numBytes 与空闲空间 memoryFree 的差值，如果 numBytesToFree 大于 0，则说明需要腾出部分 Block 所占用的内存空间。然后调用重载的 acquireMemory 方法申请获得内存。重载的 acquireMemory 方法（见代码清单 6-37）用于给 BlockId 对应的 Block 获取 Block 所需大小（即 numBytesToAcquire）的内存。当 StorageMemoryPool 内存不足时，还需要腾出其他 Block 占用的内存给当前 Block，腾出的大小由 numBytesToFree 属性指定。

代码清单6-37 获取内存

```
def acquireMemory(
    blockId: BlockId,
    numBytesToAcquire: Long,
    numBytesToFree: Long): Boolean = lock.synchronized {
  assert(numBytesToAcquire >= 0)
  assert(numBytesToFree >= 0)
  assert(memoryUsed <= poolSize)
  if (numBytesToFree > 0) { // 腾出numBytesToFree属性指定大小的空间
    memoryStore.evictBlocksToFreeSpace(Some(blockId), numBytesToFree, memoryMode)
  }
  val enoughMemory = numBytesToAcquire <= memoryFree // 判断内存是否充足
  if (enoughMemory) {
    _memoryUsed += numBytesToAcquire // 增加已经使用的内存大小
  }
  enoughMemory // 返回是否成功获得了用于存储Block的内存空间
}
```

根据代码清单 6-37，acquireMemory 方法的执行步骤如下。

1）调用 MemoryStore 的 evictBlocksToFreeSpace 方法（见代码清单 6-57），腾出 numBytesToFree 属性指定大小的空间。

2）判断内存是否充足（即 numBytesToAcquire 是否小于等于 memoryFree）。

3）如果内存充足，则将 numBytesToAcquire 增加到 _memoryUsed（即逻辑上获得了用于存储 Block 的内存空间）。

4）返回布尔值 enoughMemory，即是否成功获得了用于存储 Block 的内存空间。

2. releaseMemory

releaseMemory 方法用于释放内存，其实现如代码清单 6-38 所示。

代码清单6-38 释放内存

```
def releaseMemory(size: Long): Unit = lock.synchronized {
  if (size > _memoryUsed) {
    logWarning(s"Attempted to release $size bytes of storage " +
      s"memory when we only have ${_memoryUsed} bytes")
    _memoryUsed = 0
  } else {
    _memoryUsed -= size
  }
}
```

3. freeSpaceToShrinkPool

freeSpaceToShrinkPool 方法（见代码清单 6-39）用于释放指定大小的空间，缩小内存池的大小。

代码清单6-39 freeSpaceToShrinkPool的实现

```
def freeSpaceToShrinkPool(spaceToFree: Long): Long = lock.synchronized {
```

```
    val spaceFreedByReleasingUnusedMemory = math.min(spaceToFree, memoryFree)
    val remainingSpaceToFree = spaceToFree - spaceFreedByReleasingUnusedMemory
    if (remainingSpaceToFree > 0) {
      val spaceFreedByEviction =
        memoryStore.evictBlocksToFreeSpace(None, remainingSpaceToFree, memoryMode)
      spaceFreedByReleasingUnusedMemory + spaceFreedByEviction
    } else {
      spaceFreedByReleasingUnusedMemory
    }
  }
```

根据代码清单6-39，freeSpaceToShrinkPool方法的逻辑为缩小内存池指定的大小（即spaceToFree）。如果空闲内存（即memoryFree）大于等于spaceToFree，那么返回spaceToFree即可；否则需要腾出占用内存的部分Block，以补齐spaceToFree的大小，并返回腾出的空间与memoryFree的大小之和。由于腾出了占用内存的部分Block，所以应该减少_memoryUsed的大小，但是并未看到代码中如此实现，这是因为调用evictBlocksToFreeSpace腾出Block时，evictBlocksToFreeSpace会调用blockEviction-Handler（即BlockManager）的dropFromMemory，而BlockManager的dropFromMemory方法将会调用StorageMemoryPool的releaseMemory方法（见代码清单6-38），因此此处不再需要减少_memoryUsed的大小。

6.5.3 MemoryManager模型

有了MemoryPool模型和StorageMemoryPool的基础，我们可以来看看抽象类MemoryManager定义的内存管理器的接口规范了。了解MemoryManager也应当从其属性开始。MemoryManager的属性中既有和存储体系相关的，也有和任务执行相关的，这些属性如下。

- conf：即SparkConf。
- numCores：CPU内核数。
- onHeapStorageMemory：用于存储的堆内存大小。
- onHeapExecutionMemory：用于执行计算的堆内存大小。
- onHeapStorageMemoryPool：用于堆内存的存储内存池（StorageMemoryPool），大小由onHeapStorageMemory属性指定。
- offHeapStorageMemoryPool：堆外内存的存储内存池（StorageMemoryPool）。
- onHeapExecutionMemoryPool：堆内存的执行计算内存池（ExecutionMemoryPool），大小由onHeapExecutionMemory属性指定。
- offHeapExecutionMemoryPool：堆外内存的执行计算内存池（ExecutionMemoryPool）。
- maxOffHeapMemory：堆外内存的最大值。可以通过spark.memory.offHeap.size属性指定，默认为0。
- offHeapStorageMemory：用于存储的堆外内存大小。可以通过spark.memory.storage-

Fraction 属性（默认为 0.5）修改存储占用堆外内存的分数大小来影响 offHeapStorageMemory 的大小。offHeapStorageMemory 的计算公式为：

```
offHeapStorageMemory = maxOffHeapMemory * spark.memory.storageFraction
```

根据我们对 MemoryManager 属性的了解，可以用图 6-6 来表示 MemoryManager 管理的 4 块内存池。

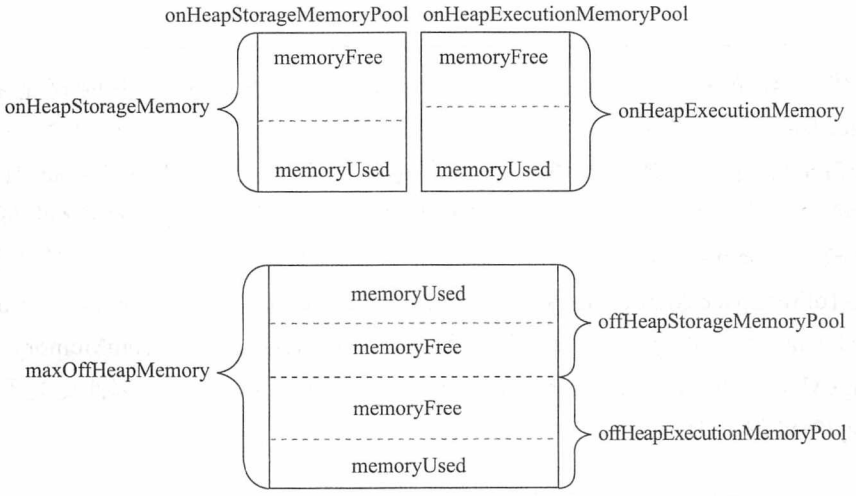


图 6-6　MemoryManager 管理的 4 块内存池

有了对 MemoryManager 管理的内存从整体上的认识，现在来看看 MemoryManager 提供了哪些方法。由于 MemoryManager 中提供的方法既有用于存储体系的，也有用于任务计算的，所以本节只介绍 MemoryManager 中提供的与存储体系有关的方法，这些方法包括如下几个。

- maxOnHeapStorageMemory：返回用于存储的最大堆内存。此方法需要子类实现。
- maxOffHeapStorageMemory：返回用于存储的最大堆外内存。此方法需要子类实现。
- setMemoryStore：给 onHeapStorageMemoryPool 和 offHeapStorageMemoryPool 设置 MemoryStore。setMemoryStore 方法（见代码清单 6-40）实际调用了 StorageMemoryPool 的 setMemoryStore 方法。

代码清单6-40　setMemoryStore的实现

```
final def setMemoryStore(store: MemoryStore): Unit = synchronized {
  onHeapStorageMemoryPool.setMemoryStore(store)
  offHeapStorageMemoryPool.setMemoryStore(store)
}
```

- acquireStorageMemory：为存储 BlockId 对应的 Block，从堆内存或堆外内存获取所需大小（即 numBytes）的内存。此方法的接口定义如下。

```
def acquireStorageMemory(blockId: BlockId, numBytes: Long, memoryMode:
  MemoryMode): Boolean
```

- acquireUnrollMemory：为展开 BlockId 对应的 Block，从堆内存或堆外内存获取所需大小（即 numBytes）的内存。此方法的接口定义如下。

```
def acquireUnrollMemory(blockId: BlockId, numBytes: Long, memoryMode:
  MemoryMode): Boolean
```

- releaseStorageMemory：从堆内存或堆外内存释放指定大小（即 numBytes）的内存。此方法的实现如代码清单 6-41 所示。

代码清单6-41 releaseStorageMemory的实现

```
def releaseStorageMemory(numBytes: Long, memoryMode: MemoryMode): Unit = synchronized {
  memoryMode match {
    case MemoryMode.ON_HEAP => onHeapStorageMemoryPool.releaseMemory(numBytes)
    case MemoryMode.OFF_HEAP => offHeapStorageMemoryPool.releaseMemory(numBytes)
  }
}
```

- releaseAllStorageMemory：从堆内存及堆外内存释放所有内存。此方法的实现如代码清单 6-42 所示。

代码清单6-42 releaseAllStorageMemory的实现

```
final def releaseAllStorageMemory(): Unit = synchronized {
  onHeapStorageMemoryPool.releaseAllMemory()
  offHeapStorageMemoryPool.releaseAllMemory()
}
```

- releaseUnrollMemory：释放指定大小（即 numBytes）的展开内存。此方法的实现如代码清单 6-43 所示。

代码清单6-43 releaseUnrollMemory的实现

```
final def releaseUnrollMemory(numBytes: Long, memoryMode: MemoryMode): Unit =
    synchronized {
  releaseStorageMemory(numBytes, memoryMode)
}
```

- storageMemoryUsed：onHeapStorageMemoryPool 与 offHeapStorageMemoryPool 中一共占用的存储内存。此方法的实现如代码清单 6-44 所示。

代码清单6-44 storageMemoryUsed的实现

```
final def storageMemoryUsed: Long = synchronized {
  onHeapStorageMemoryPool.memoryUsed + offHeapStorageMemoryPool.memoryUsed
}
```

MemoryManager 有两个子类，分别是 StaticMemoryManager 和 UnifiedMemoryManager。StaticMemoryManager 保留了早期 Spark 版本遗留下来的静态内存管理机制，也不存在堆外内存的模型。在静态内存管理机制下，Spark 应用程序在运行期的存储内存和执行内存的大小均为固定的。从 Spark 1.6.0 版本开始，以 UnifiedMemoryManager 作为默认的内存管理

器。UnifiedMemoryManager 提供了统一的内存管理机制，即 Spark 应用程序在运行期的存储内存和执行内存将共享统一的内存空间，可以动态调节两块内存的空间大小，所以下一节将着重介绍 UnifiedMemoryManager。

6.5.4 UnifiedMemoryManager 详解

UnifiedMemoryManager 在 MemoryManager 的内存模型之上，将计算内存和存储内存之间的边界修改为“软”边界，即任何一方可以向另一方借用空闲的内存。为了对 UnifiedMemoryManager 的实现原理有更深入的理解，还是先掌握 UnifiedMemoryManager 的成员属性。

1）conf：即 SparkConf。此构造器属性将用于父类 MemoryManager 的构造器属性——conf。

2）maxHeapMemory：最大堆内存。大小为系统可用内存与 spark.memory.fraction 属性值（默认为 0.6）的乘积。

3）onHeapStorageRegionSize：用于存储的堆内存大小。此构造器属性将用于父类 MemoryManager 的构造器属性——onHeapStorageMemory。由于 UnifiedMemoryManager 的构造器属性中没有 onHeapExecutionMemory，所以 maxHeapMemory 与 onHeapStorageRegionSize 的差值就作为父类 MemoryManager 的构造器属性——onHeapExecutionMemory。

4）numCores：CPU 内核数。此构造器属性将用于父类 MemoryManager 的构造器属性——numCores。

有了对 UnifiedMemoryManager 属性的了解，我们针对 UnifiedMemoryManager 对图 6-6 中的四块内存池进行调整，如图 6-7 所示。

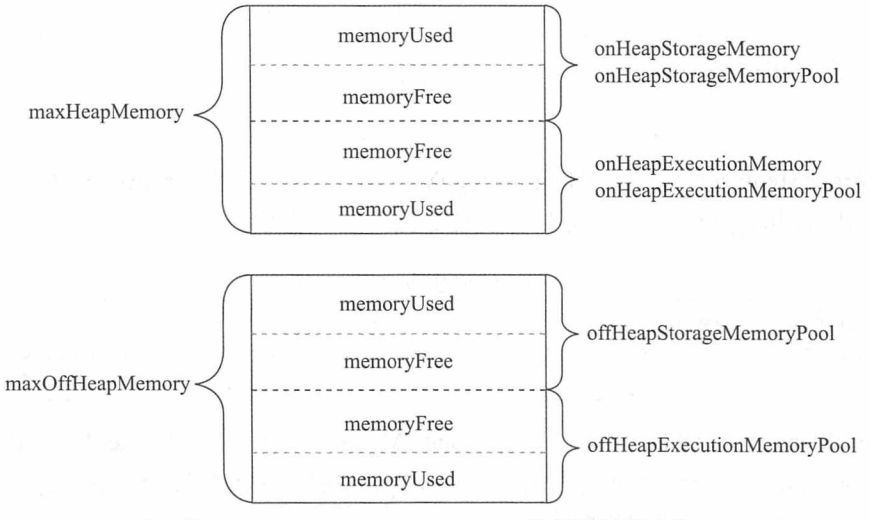


图 6-7 UnifiedMemoryManager 管理的四块内存

图 6-7 将图 6-6 中毫不相干的 onHeapStorageMemoryPool 和 onHeapExecutionMemoryPool 合在了一起，将堆内存作为一个整体看待。而且 onHeapStorageMemoryPool 与 onHeapExecutionMemoryPool 之间，offHeapStorageMemoryPool 与 offHeapExecutionMemoryPool 之间的实线也调整为虚线，表示它们之间都是“软”边界。存储方或计算方的空闲空间（即 memoryFree 表示的区域）都可以借给另一方使用。

有了前面的铺垫，现在来看看 UnifiedMemoryManager 实现的继承自父类 MemoryManager 的方法。由于 UnifiedMemoryManager 中既有存储相关的方法，也有执行内存相关的方法，所以本节只选择存储相关的方法进行介绍，执行内存相关的方法留到第 8 章进行介绍。

1）maxOnHeapStorageMemory：返回用于存储的最大堆内存。此方法的实现如代码清单 6-45 所示。

代码清单6-45 maxOnHeapStorageMemory的实现

```
override def maxOnHeapStorageMemory: Long = synchronized {
  maxHeapMemory - onHeapExecutionMemoryPool.memoryUsed
}
```

根据代码清单 6-45，你可能对 maxOnHeapStorageMemory 的实现提出质疑：“maxOnHeapStorageMemory 不是应该等于 onHeapStorageMemory 吗？”这说明你的确理解了 MemoryManager 的内存模型。在本节开头曾经说过，计算内存和存储内存之间的“软”边界和互借内存的概念，所以凡是计算内存没有使用的或空闲的内存，都有可能被借用，作为当前内存管理器用于存储的部分。

2）maxOffHeapStorageMemory：返回用于存储的最大堆外内存。此方法的实现如代码清单 6-46。

代码清单6-46 maxOffHeapStorageMemory的实现

```
override def maxOffHeapStorageMemory: Long = synchronized {
  maxOffHeapMemory - offHeapExecutionMemoryPool.memoryUsed
}
```

根据代码清单 6-46，我们知道存储方和计算方不光是堆内存可以互借，堆外内存的空间也可以互借。

3）acquireStorageMemory：为存储 BlockId 对应的 Block，从堆内存或堆外内存获取所需大小（即 numBytes）的内存。此方法的实现如代码清单 6-47 所示。

代码清单6-47 acquireStorageMemory的实现

```
override def acquireStorageMemory(
    blockId: BlockId,
    numBytes: Long,
    memoryMode: MemoryMode): Boolean = synchronized {
  assertInvariants()
```

```
    assert(numBytes >= 0)
    // 获取内存模式对应的计算内存池、存储内存池和可以存储的最大空间
    val (executionPool, storagePool, maxMemory) = memoryMode match {
      case MemoryMode.ON_HEAP => (
        onHeapExecutionMemoryPool,
        onHeapStorageMemoryPool,
        maxOnHeapStorageMemory)
      case MemoryMode.OFF_HEAP => (
        offHeapExecutionMemoryPool,
        offHeapStorageMemoryPool,
        maxOffHeapMemory)
    }
    if (numBytes > maxMemory) { // numBytes不能大于可以存储的最大空间
      logInfo(s"Will not store $blockId as the required space ($numBytes bytes)
        exceeds our " +
        s"memory limit ($maxMemory bytes)")
      return false
    }
    if (numBytes > storagePool.memoryFree) { // 存储内存池空间不足，到计算内存池中去借用
      空间
      val memoryBorrowedFromExecution = Math.min(executionPool.memoryFree, numBytes)
      executionPool.decrementPoolSize(memoryBorrowedFromExecution)
      storagePool.incrementPoolSize(memoryBorrowedFromExecution)
    }
     storagePool.acquireMemory(blockId, numBytes) //从存储内存池获得存储BlockId对应的
  Block所需的空间
  }
```

根据代码清单 6-47，acquireStorageMemory 方法的执行步骤如下。

① 根据内存模式，获取此内存模式的计算内存池、存储内存池和可以存储的最大空间。

② 对要获得的存储大小进行校验，即 numBytes 不能大于可以存储的最大空间。

③ 如果要获得的存储大小比存储内存池的空闲空间要大，那么就到计算内存池中去借用空间。借用的空间取 numBytes 和计算内存池的空闲空间的最小值。

④ 从存储内存池获得存储 BlockId 对应的 Block 所需的空间。

4）acquireUnrollMemory：为展开 BlockId 对应的 Block，从堆内存或堆外内存获取所需大小（即 numBytes）的内存。此方法实际代理调用了 acquireStorageMemory，没有任何额外的逻辑，故省略其代码实现。

6.6 内存存储 MemoryStore

MemoryStore 负责将 Block 存储到内存。Spark 通过将广播数据、RDD、Shuffle 数据存储到内存，减少了对磁盘 I/O 的依赖，提高了程序的读写效率。由于 MemoryStore 依赖于 MemoryManager 的服务，所以我们在上一节先介绍了 MemoryManager。下面我们将首先分析 MemoryStore 的内存模型，然后介绍 MemoryStore 提供的方法。

6.6.1 MemoryStore 的内存模型

Block 在内存中以什么形式存在呢？是将文件直接缓存到内存？Spark 将内存中的 Block 抽象为特质 MemoryEntry，其定义如下。

```
private sealed trait MemoryEntry[T] {
  def size: Long
  def memoryMode: MemoryMode
  def classTag: ClassTag[T]
}
```

根据上面的代码可以看到，MemoryEntry 提供了三个接口方法。

- size：当前 Block 的大小。
- memoryMode：Block 存入内存的内存模式。
- classTag：Block 的类型标记。

MemoryEntry 有两个实现类，它们的实现如下。

```
private case class DeserializedMemoryEntry[T](
    value: Array[T],
    size: Long,
    classTag: ClassTag[T]) extends MemoryEntry[T] {
    val memoryMode: MemoryMode = MemoryMode.ON_HEAP
  }
  private case class SerializedMemoryEntry[T](
    buffer: ChunkedByteBuffer,
    memoryMode: MemoryMode,
    classTag: ClassTag[T]) extends MemoryEntry[T] {
  def size: Long = buffer.size
}
```

DeserializedMemoryEntry 表示反序列化后的 MemoryEntry，而 SerializedMemoryEntry 表示序列化后的 MemoryEntry。

为了了解 MemoryStore，我们先来看看 MemoryStore 的属性信息。

- conf：即 SparkConf。
- blockInfoManager：即 Block 信息管理器 BlockInfoManager。
- serializerManager：即序列化管理器 SerializerManager。
- memoryManager：即内存管理器 MemoryManager。MemoryStore 存储 Block，使用的就是 MemoryManager 内的 maxOnHeapStorageMemory 和 maxOffHeapStorageMemory 两块内存池。
- blockEvictionHandler：Block 驱逐处理器。blockEvictionHandler 用于将 Block 从内存中驱逐出去。blockEvictionHandler 的类型是 BlockEvictionHandler，BlockEvictionHandler 定义了将对象从内存中移除的接口，BlockEvictionHandler 的定义如下。

```
private[storage] trait BlockEvictionHandler {
  private[storage] def dropFromMemory[T: ClassTag](
    blockId: BlockId,
```

```
    data: () => Either[Array[T], ChunkedByteBuffer]): StorageLevel
}
```

BlockManager 实现了特质 BlockEvictionHandler，并重写了 dropFromMemory 方法，BlockManager 在构造 MemoryStore 时，将自身的引用作为 blockEvictionHandler 参数传递给 MemoryStore 的构造器，因而 BlockEvictionHandler 就是 BlockManager。BlockManager 及 BlockManager 重写的 dropFromMemory 方法的实现都将在 6.7 节详细介绍。

- entries：内存中的 BlockId 与 MemoryEntry（Block 的内存形式）之间映射关系的缓存。
- onHeapUnrollMemoryMap：任务尝试线程的标识 TaskAttemptId 与任务尝试线程在堆内存展开的所有 Block 占用的内存大小之和之间的映射关系。
- offHeapUnrollMemoryMap：任务尝试线程的标识 TaskAttemptId 与任务尝试线程在堆外内存展开的所有 Block 占用的内存大小之和之间的映射关系。
- unrollMemoryThreshold：用来展开任何 Block 之前，初始请求的内存大小，可以修改属性 spark.storage.unrollMemoryThreshold（默认为 1MB）改变大小。

MemoryStore 除了以上属性外，还有一些方法对 MemoryStore 的模型提供了概念上的描述。

- maxMemory：MemoryStore 用于存储 Block 的最大内存，其实质为 MemoryManager 的 maxOnHeapStorageMemory 和 maxOffHeapStorageMemory 之和。如果 MemoryManager 为 StaticMemoryManager，那么 maxMemory 的大小是固定的。如果 MemoryManager 为 UnifiedMemoryManager，那么 maxMemory 的大小是动态变化的。
- memoryUsed：MemoryStore 中已经使用的内存大小。其实质为 MemoryManager 中 onHeapStorageMemoryPool 已经使用的大小和 offHeapStorageMemoryPool 已经使用的大小之和。
- currentUnrollMemory：MemoryStore 用于展开 Block 使用的内存大小。其实质为 onHeapUnrollMemoryMap 中的所有用于展开 Block 所占用的内存大小与 offHeapUnrollMemoryMap 中的所有用于展开 Block 所占用的内存大小之和。
- blocksMemoryUsed：MemoryStore 用于存储 Block（即 MemoryEntry）使用的内存大小，即 memoryUsed 与 currentUnrollMemory 的差值。
- currentUnrollMemoryForThisTask：当前的任务尝试线程用于展开 Block 所占用的内存。即 onHeapUnrollMemoryMap 中缓存的当前任务尝试线程对应的占用大小与 offHeapUnrollMemoryMap 中缓存的当前的任务尝试线程对应的占用大小之和。
- numTasksUnrolling：当前使用 MemoryStore 展开 Block 的任务的数量。其实质为 onHeapUnrollMemoryMap 的键集合与 offHeapUnrollMemoryMap 的键集合的并集。

有了对这些 MemoryStore 中成员的了解，我们可以来看看 MemoryStore 的内存模型。

MemoryStore 相比于 MemoryManager，提供了一种宏观的内存模型，MemoryManager 模型的堆内存和堆外内存在 MemoryStore 的内存模型中是透明的，UnifiedMemoryManager 中存储内存与计算内存的“软”边界在 MemoryStore 的内存模型中也是透明的，读者应该时刻记着这些不同点。MemoryStore 的内存模型如图 6-8 所示。

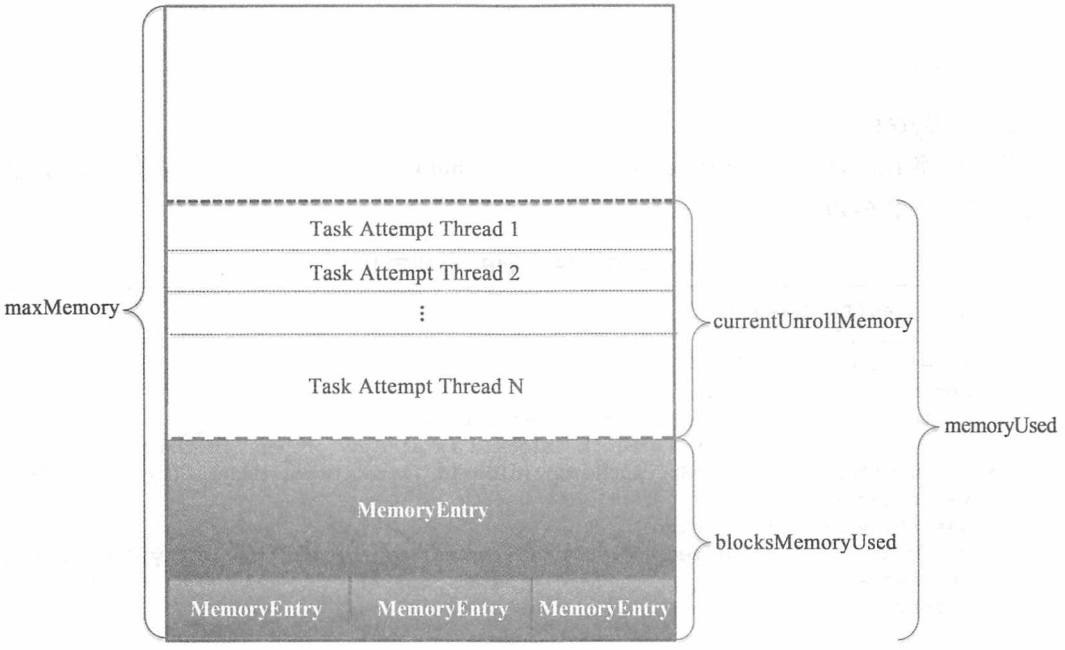


图 6-8 MemoryStore 的内存模型

从图中看出，整个 MemoryStore 的存储分为三块：一块是 MemoryStore 的 entries 属性持有的很多 MemoryEntry 所占据的内存 blocksMemoryUsed；一块是 onHeapUnrollMemoryMap 或 offHeapUnrollMemoryMap 中使用展开方式占用的内存 currentUnrollMemory。展开 Block 的行为类似于人们生活中的“占座”，一间教室里有些座位有人，有些则空着。在座位上放一本书表示有人正在使用，那么别人就不会坐这些座位。这可以防止在你需要座位的时候，却发现已经没有了位置。这样可以防止在向内存真正写入数据时，内存不足发生溢出。blocksMemoryUsed 和 currentUnrollMemory 的空间之和是已经使用的空间，用 memoryUsed 表示。还有一块内存没有任何标记，表示未使用。

6.6.2 MemoryStore 提供的方法

MemoryStore 提供了很多方法，便于对 Block 数据的存储和读取。MemoryStore 提供的方法如下。

1. getSize

此方法用于获取 BlockId 对应 MemoryEntry（即 Block 的内存形式）所占用的大小。此

方法的实现如代码清单 6-48 所示。

代码清单6-48 getSize的实现

```
def getSize(blockId: BlockId): Long = {
  entries.synchronized {
    entries.get(blockId).size
  }
}
```

2. putBytes

此方法将 BlockId 对应的 Block（已经封装为 ChunkedByteBuffer）写入内存。此方法的实现如代码清单 6-49 所示。

代码清单6-49 putBytes的实现

```
def putBytes[T: ClassTag](
    blockId: BlockId,
    size: Long,
    memoryMode: MemoryMode,
    _bytes: () => ChunkedByteBuffer): Boolean = {
  require(!contains(blockId), s"Block $blockId is already present in the MemoryStore")
  if (memoryManager.acquireStorageMemory(blockId, size, memoryMode)) { // 获取逻辑内存
    val bytes = _bytes() // 获取Block的数据
    assert(bytes.size == size)
    val entry = new SerializedMemoryEntry[T](bytes, memoryMode, implicitly[ClassTag[T]])
    entries.synchronized {
      entries.put(blockId, entry) // 将Block数据写入内存
    }
    logInfo("Block %s stored as bytes in memory (estimated size %s, free %s)".format(
      blockId, Utils.bytesToString(size), Utils.bytesToString(maxMemory - blocksMemoryUsed)))
    true
  } else {
    false
  }
}
```

根据代码清单 6-49，putBytes 的执行步骤如下。

1）从 MemoryManager 中获取用于存储 BlockId 对应的 Block 的逻辑内存。如果获取失败则返回 false，否则进入下一步。

2）调用 _bytes 函数，获取 Block 的数据，即 ChunkedByteBuffer。

3）创建 Block 对应的 SerializedMemoryEntry。

4）将 SerializedMemoryEntry 放入 entries 缓存。

5）返回 true。

3. reserveUnrollMemoryForThisTask

此方法用于为展开尝试执行任务给定的 Block，保留指定内存模式上指定大小的内存。reserveUnrollMemoryForThisTask 的实现如代码清单 6-50 所示。

书籍是人类进步的阶梯

代码清单6-50 reserveUnrollMemoryForThisTask的实现

```
def reserveUnrollMemoryForThisTask(
    blockId: BlockId,
    memory: Long,
    memoryMode: MemoryMode): Boolean = {
  memoryManager.synchronized {
    val success = memoryManager.acquireUnrollMemory(blockId, memory, memoryMode)
       // 获取展开内存
    if (success) {
      val taskAttemptId = currentTaskAttemptId() // 更新taskAttemptId与展开内存大小
         之间的映射关系
      val unrollMemoryMap = memoryMode match {
        case MemoryMode.ON_HEAP => onHeapUnrollMemoryMap
        case MemoryMode.OFF_HEAP => offHeapUnrollMemoryMap
      }
      unrollMemoryMap(taskAttemptId) = unrollMemoryMap.getOrElse(taskAttemptId,
          0L) + memory
    }
    success // 返回获取成功或失败的状态
  }
}
```

根据代码清单 6-50，reserveUnrollMemoryForThisTask 的执行步骤如下。

1）调用 MemoryManager 的 acquireUnrollMemory 方法获取展开内存。

2）如果获取内存成功，则更新 taskAttemptId 与任务尝试线程在堆内存或堆外内存展开的所有 Block 占用的内存大小之和之间的映射关系。

3）返回获取成功或失败的状态。

4. releaseUnrollMemoryForThisTask

此方法用于释放任务尝试线程占用的内存，releaseUnrollMemoryForThisTask 的实现如代码清单 6-51 所示。

代码清单6-51 releaseUnrollMemoryForThisTask的实现

```
def releaseUnrollMemoryForThisTask(memoryMode: MemoryMode, memory: Long = Long.
   MaxValue): Unit = {
  val taskAttemptId = currentTaskAttemptId()
  memoryManager.synchronized {
    val unrollMemoryMap = memoryMode match {
      case MemoryMode.ON_HEAP => onHeapUnrollMemoryMap
      case MemoryMode.OFF_HEAP => offHeapUnrollMemoryMap
    }
    if (unrollMemoryMap.contains(taskAttemptId)) {
      val memoryToRelease = math.min(memory, unrollMemoryMap(taskAttemptId)) //计
         算要释放的内存
      if (memoryToRelease > 0) { // 释放展开内存
        unrollMemoryMap(taskAttemptId) -= memoryToRelease
        memoryManager.releaseUnrollMemory(memoryToRelease, memoryMode)
      }
      if (unrollMemoryMap(taskAttemptId) == 0) {
        unrollMemoryMap.remove(taskAttemptId) // 清除taskAttemptId与展开内存大小之间
```

书籍是人类进步的阶梯

```
                的映射关系
            }
        }
    }
}
```

根据代码清单 6-51，releaseUnrollMemoryForThisTask 的执行步骤如下。

1）计算实际要释放的内存大小，此大小为指定要释放的大小和任务尝试线程在堆内存或堆外内存实际占有的内存大小之和之间的最小值。

2）更新 taskAttemptId 与任务尝试线程在堆内存或堆外内存展开的所有 Block 占用的内存大小之和之间的映射关系。

3）调用 MemoryManager 的 releaseUnrollMemory 方法释放内存。

4）如果任务尝试线程在堆内存或堆外内存展开的所有 Block 占用的内存大小之和为 0，则清除此 taskAttemptId 与任务尝试线程在堆内存或堆外内存展开的所有 Block 占用的内存大小之和之间的映射关系。

5. putIteratorAsValues

此方法将 BlockId 对应的 Block（已经转换为 Iterator）写入内存。有时候放入内存的 Block 很大，所以一次性将此对象写入内存可能将引发 OOM（即 java.lang.OutOfMemoryError）异常。为了避免这种情况的发生，首先需要将 Block 转换为 Iterator，然后渐进式地展开此 Iterator，并且周期性地检查是否有足够的展开内存。此方法涉及很多变量，为了便于理解，这里先解释这些变量的含义，然后再分析方法实现。

putIteratorAsValues 中的变量包括以下几个。

- elementsUnrolled：已经展开的元素数量。
- keepUnrolling：MemoryStore 是否仍然有足够的内存，以便于继续展开 Block（即 Iterator）。
- initialMemoryThreshold：即 unrollMemoryThreshold。用来展开任何 Block 之前，初始请求的内存大小，可以修改属性 spark.storage.unrollMemoryThreshold（默认为 1MB）改变大小。
- memoryCheckPeriod：检查内存是否有足够的阀值，此值固定为 16。字面上有周期的含义，但是此周期并非指时间，而是已经展开的元素的数量 elementsUnrolled。
- memoryThreshold：当前任务用于展开 Block 所保留的内存。
- memoryGrowthFactor：展开内存不充足时，请求增长的因子。此值固定为 1.5。
- unrollMemoryUsedByThisBlock：Block 已经使用的展开内存大小，初始大小为 initialMemoryThreshold。
- vector：用于追踪 Block（即 Iterator）每次迭代的数据。类型为 SizeTrackingVector。

有了对 putIteratorAsValues 方法中各个变量的掌握，理解 putIteratorAsValues 的实现将更加容易。putIteratorAsValues 的实现如代码清单 6-52 所示。

书籍是人类进步的阶梯

代码清单6-52 putIteratorAsValues的实现

```
private[storage] def putIteratorAsValues[T](
    blockId: BlockId,
    values: Iterator[T],
    classTag: ClassTag[T]): Either[PartiallyUnrolledIterator[T], Long] = {
  // 忽略声明变量的代码
  // 不断迭代读取Iterator中的数据，将数据放入追踪器vector中
  while (values.hasNext && keepUnrolling) {
    vector += values.next()
    if (elementsUnrolled % memoryCheckPeriod == 0) { // 周期性地检查
      val currentSize = vector.estimateSize()
      if (currentSize >= memoryThreshold) {
        val amountToRequest = (currentSize * memoryGrowthFactor - memory-
           Threshold).toLong
        keepUnrolling =
          reserveUnrollMemoryForThisTask(blockId, amountToRequest, MemoryMode.ON_HEAP)
        if (keepUnrolling) {
          unrollMemoryUsedByThisBlock += amountToRequest
        }
        memoryThreshold += amountToRequest
      }
    }
    elementsUnrolled += 1
  }

  if (keepUnrolling) { // 申请到足够多的展开内存，将数据写入内存
    val arrayValues = vector.toArray
    vector = null
    val entry =
      new DeserializedMemoryEntry[T](arrayValues, SizeEstimator.estimate(arrayValues),
         classTag)
    val size = entry.size
    def transferUnrollToStorage(amount: Long): Unit = { // 将展开Block的内存转换为存
                                                        // 储Block的内存
      memoryManager.synchronized {
        releaseUnrollMemoryForThisTask(MemoryMode.ON_HEAP, amount)
        val success = memoryManager.acquireStorageMemory(blockId, amount,
           MemoryMode.ON_HEAP)
        assert(success, "transferring unroll memory to storage memory failed")
      }
    }
    val enoughStorageMemory = {
      if (unrollMemoryUsedByThisBlock <= size) {
        val acquiredExtra =
          memoryManager.acquireStorageMemory(
            blockId, size - unrollMemoryUsedByThisBlock, MemoryMode.ON_HEAP)
        if (acquiredExtra) {
          transferUnrollToStorage(unrollMemoryUsedByThisBlock)
        }
        acquiredExtra
      } else { // 当unrollMemoryUsedByThisBlock > size，归还多余的展开内存空间
        val excessUnrollMemory = unrollMemoryUsedByThisBlock - size
        releaseUnrollMemoryForThisTask(MemoryMode.ON_HEAP, excessUnrollMemory)
        transferUnrollToStorage(size)
        true
```

书籍是人类进步的阶梯

```
        }
      }
      if (enoughStorageMemory) {
        entries.synchronized {
          entries.put(blockId, entry)
        }
        logInfo("Block %s stored as values in memory (estimated size %s, free %s)".
            format(
          blockId, Utils.bytesToString(size), Utils.bytesToString(maxMemory -
              blocksMemoryUsed)))
        Right(size)
      } else {
        assert(currentUnrollMemoryForThisTask >= unrollMemoryUsedByThisBlock,
          "released too much unroll memory")
        Left(new PartiallyUnrolledIterator(
          this,
          MemoryMode.ON_HEAP,
          unrollMemoryUsedByThisBlock,
          unrolled = arrayValues.toIterator,
          rest = Iterator.empty))
      }
    } else {
      logUnrollFailureMessage(blockId, vector.estimateSize())
      Left(new PartiallyUnrolledIterator(
        this,
        MemoryMode.ON_HEAP,
        unrollMemoryUsedByThisBlock,
        unrolled = vector.iterator,
        rest = values))
    }
  }
```

根据代码清单 6-52，putIteratorAsValues 方法的执行步骤如下。

1）不断迭代读取 Iterator 中的数据，将数据放入追踪器 vector 中，并周期性地检查 vector 中所有数据的估算大小 currentSize 是否已经超过了 memoryThreshold。当发现 currentSize 超过 memoryThreshold，则为当前任务请求新的保留内存（内存大小的计算公式为：currentSize * memoryGrowthFactor – memoryThreshold）。在堆上成功申请到足够的内存后，需要更新 unrollMemoryUsedByThisBlock 和 memoryThreshold 的大小。

2）如果展开 Iterator 中所有的数据后，keepUnrolling 为 true，则说明已经为 Block 申请到足够多的保留内存，接下来的处理步骤如下。

① 将 vector 中的数据封装为 DeserializedMemoryEntry，并重新估算 vector 的大小 size。

② 如果 unrollMemoryUsedByThisBlock 大于 size，说明用于展开的内存过多，需要向 MemoryManager 归还多余的空间。归还的内存大小为 unrollMemoryUsedByThisBlock – size。之后调用 transferUnrollToStorage 方法将展开 Block 占用的内存转换为用于存储 Block 的内存，此转换过程是原子的。

③ 如果有足够的内存存储 Block，则将 BlockId 与 DeserializedMemoryEntry 的映射关系放入 entries 并返回 Right（size）。

书籍是人类进步的阶梯

④ 如果没有足够的内存存储 Block，则创建 PartiallyUnrolledIterator 并返回 Left。

3）如果展开 Iterator 中所有的数据后，keepUnrolling 为 false，说明没有为 Block 申请到足够多的保留内存，此时将创建 PartiallyUnrolledIterator 并返回 Left。

6. putIteratorAsBytes

此方法以序列化后的字节数组方式，将 BlockId 对应的 Block（已经转换为 Iterator）写入内存。由于此方法的实现与 putIteratorAsValues 类似，所以留给感兴趣的读者自行阅读理解。

7. getBytes

此方法从内存中读取 BlockId 对应的 Block（已经封装为 ChunkedByteBuffer）。此方法的实现如代码清单 6-53 所示。

代码清单6-53 getBytes的实现

```
def getBytes(blockId: BlockId): Option[ChunkedByteBuffer] = {
  val entry = entries.synchronized { entries.get(blockId) }
  entry match {
    case null => None
    case e: DeserializedMemoryEntry[_] =>
      throw new IllegalArgumentException("should only call getBytes on serialized blocks")
    case SerializedMemoryEntry(bytes, _, _) => Some(bytes)
  }
}
```

根据代码清单 6-53，我们知道 getBytes 只能获取序列化的 Block。

8. getValues

此方法用于从内存中读取 BlockId 对应的 Block（已经封装为 Iterator）。此方法的实现如代码清单 6-54 所示。

代码清单6-54 getValues的实现

```
def getValues(blockId: BlockId): Option[Iterator[_]] = {
  val entry = entries.synchronized { entries.get(blockId) }
  entry match {
    case null => None
    case e: SerializedMemoryEntry[_] =>
      throw new IllegalArgumentException("should only call getValues on
          deserialized blocks")
    case DeserializedMemoryEntry(values, _, _) =>
      val x = Some(values)
      x.map(_.iterator)
  }
}
```

根据代码清单 6-54，我们知道 getValues 只能获取没有序列化的 Block。

9. remove

此方法用于从内存中移除 BlockId 对应的 Block，remove 的实现如代码清单 6-55 所示。

书籍是人类进步的阶梯

代码清单6-55 移除Block

```
def remove(blockId: BlockId): Boolean = memoryManager.synchronized {
  val entry = entries.synchronized {
    entries.remove(blockId) // 将Block从内存移除
  }
  if (entry != null) {
    entry match {
      case SerializedMemoryEntry(buffer, _, _) => buffer.dispose()
      case _ =>
    }
    // 释放逻辑的存储内存
    memoryManager.releaseStorageMemory(entry.size, entry.memoryMode)
    logDebug(s"Block $blockId of size ${entry.size} dropped " +
      s"from memory (free ${maxMemory - blocksMemoryUsed})")
    true // 移除成功
  } else {
    false // 移除失败
  }
}
```

根据代码清单 6-55，remove 的执行步骤如下。

1）将 BlockId 对应的 MemoryEntry 从 entries 中移除。如果 entries 中存在 BlockId 对应的 MemoryEntry，则进入第 2）步，否则返回 false。

2）如果 MemoryEntry 是 SerializedMemoryEntry，则还要将对应的 ChunkedByteBuffer 清理。

3）调用 MemoryManager 的 releaseStorageMemory 方法，释放使用的存储内存。

4）返回 true。

10. clear

此方法用于清空 MemoryStore，实现如代码清单 6-56 所示。

代码清单6-56 清空MemoryStore

```
def clear(): Unit = memoryManager.synchronized {
  entries.synchronized {
    entries.clear()
  }
  onHeapUnrollMemoryMap.clear()
  offHeapUnrollMemoryMap.clear()
  memoryManager.releaseAllStorageMemory()
  logInfo("MemoryStore cleared")
}
```

11. evictBlocksToFreeSpace

此方法用于驱逐 Block，以便释放一些空间来存储新的 Block，具体实现如代码清单 6-57 所示。

代码清单6-57 驱逐Block

```
private[spark] def evictBlocksToFreeSpace(
    blockId: Option[BlockId],
    space: Long,
    memoryMode: MemoryMode): Long = {
  assert(space > 0)
  memoryManager.synchronized {
    var freedMemory = 0L
    val rddToAdd = blockId.flatMap(getRddId)
    val selectedBlocks = new ArrayBuffer[BlockId]
    def blockIsEvictable(blockId: BlockId, entry: MemoryEntry[_]): Boolean = {
      entry.memoryMode == memoryMode && (rddToAdd.isEmpty || rddToAdd !=
          getRddId(blockId))
    }
    entries.synchronized {
      val iterator = entries.entrySet().iterator()
      while (freedMemory < space && iterator.hasNext) { // 选择符合驱逐条件的Block
        val pair = iterator.next()
        val blockId = pair.getKey
        val entry = pair.getValue
        if (blockIsEvictable(blockId, entry)) {
          if (blockInfoManager.lockForWriting(blockId, blocking = false).isDe-
              fined) {
            selectedBlocks += blockId
            freedMemory += pair.getValue.size
          }
        }
      }
    }

    def dropBlock[T](blockId: BlockId, entry: MemoryEntry[T]): Unit = {
      val data = entry match {
        case DeserializedMemoryEntry(values, _, _) => Left(values)
        case SerializedMemoryEntry(buffer, _, _) => Right(buffer)
      }
      val newEffectiveStorageLevel =
        blockEvictionHandler.dropFromMemory(blockId, () => data)(entry.classTag)
      if (newEffectiveStorageLevel.isValid) {
        blockInfoManager.unlock(blockId)
      } else {
        blockInfoManager.removeBlock(blockId)
      }
    }

    if (freedMemory >= space) { // 通过驱逐可以为存储Block提供足够的空间，则进行驱逐
      logInfo(s"${selectedBlocks.size} blocks selected for dropping " +
        s"(${Utils.bytesToString(freedMemory)} bytes)")
      for (blockId <- selectedBlocks) {
        val entry = entries.synchronized { entries.get(blockId) }
        if (entry != null) {
          dropBlock(blockId, entry)
        }
      }
      logInfo(s"After dropping ${selectedBlocks.size} blocks, " +
        s"free memory is ${Utils.bytesToString(maxMemory - blocksMemoryUsed)}")
```

```
        freedMemory
      } else { // 通过驱逐不能为存储Block提供足够的空间，则释放原本准备要驱逐的各个Block的写锁
        blockId.foreach { id =>
          logInfo(s"Will not store $id")
        }
        selectedBlocks.foreach { id =>
          blockInfoManager.unlock(id)
        }
        0L
      }
    }
  }
```

根据代码清单 6-57，evictBlocksToFreeSpace 中定义了一些局部变量。

- blockId：要存储的 Block 的 BlockId。
- space：需要驱逐 Block 所腾出的内存大小。
- memoryMode：存储 Block 所需的内存模式。
- freedMemory：已经释放的内存大小。
- rddToAdd：将要添加的 RDD 的 RDDBlockId 标记。rddToAdd 实际是通过对 BlockId 应用 getRddId 方法得到的。getRddId 的实现如下。

```
private def getRddId(blockId: BlockId): Option[Int] = {
  blockId.asRDDId.map(_.rddId)
}
```

上述代码说明首先调用了 BlockId 的 asRDDId 方法（见代码清单 6-3），将 BlockId 转换为 RDDBlockId，然后获取 RDDBlockId 的 rddId 属性。

- selectedBlocks：已经选择的用于驱逐的 Block 的 BlockId 的数组。

有了对以上变量的理解，现在来看看 evictBlocksToFreeSpace 的执行步骤。

1）当 freedMemory 小于 space 时，不断迭代遍历 iterator。对于每个 entries 中的 BlockId 和 MemoryEntry，首先找出其中符合条件的 Block（只有符合条件的 Block 才会被驱逐），然后获取 Block 的写锁，最后将此 Block 的 BlockId 放入 selectedBlocks 并且将 freedMemory 增加 Block 的大小。哪些 Block 将符合条件呢？同时满足以下两个条件的。

① MemoryEntry 的内存模式与所需的内存模式一致。

② BlockId 对应的 Block 不是 RDD，或者 BlockId 与 blockId 不是同一个 RDD。

2）经过了第 1）步的处理，如果 freedMemory 大于等于 space，这说明通过驱逐一定数量的 Block，已经为存储 blockId 对应的 Block 腾出了足够的内存空间，此时需要遍历 selectedBlocks 中的每个 BlockId，并移除每个 BlockId 对应的 Block。如果 Block 从内存中迁移到其他存储（如 DiskStore）中，那么需要调用 BlockInfoManager 的 unlock 释放当前任务尝试线程获取的被迁移 Block 的写锁。如果 Block 从存储体系中彻底移除了，那么需要调用 BlockInfoManager 的 removeBlock 方法删除被迁移 Block 的信息。

3）经过了第 1）步的处理，如果 freedMemory 小于 space，这说明即便驱逐内存中所

有符合条件的 Block，腾出的空间也不足以存储 blockId 对应的 Block，此时需要当前任务尝试线程释放 selectedBlocks 中每个 BlockId 对应的 Block 的写锁。


注意 有些读者可能会发现对于每个符合条件的 Block，都通过调用 BlockInfoManager 的 lockForWriting 方法获取 Block 的写锁。这是因为不应该对有线程正在读取或写入的 Block 进行驱逐，只能驱逐那些没有被其他线程读或写的 Block。

12. contains

此方法用于判断本地 MemoryStore 中是否包含给定 BlockId 所对应的 Block 文件。contains 的实现如代码清单 6-58 所示。

代码清单6-58 Block文件是否存在

```
def contains(blockId: BlockId): Boolean = {
  entries.synchronized { entries.containsKey(blockId) }
}
```

6.7 块管理器 BlockManager

BlockManager 运行在每个节点上（包括 Driver 和 Executor），提供对本地或远端节点上的内存、磁盘及堆外内存中 Block 的管理。存储体系从狭义上来说指的就是 BlockManager，从广义上来说，则包括整个 Spark 集群中的各个 BlockManager、BlockInfoManager、DiskBlockManager、DiskStore、MemoryManager、MemoryStore、对集群中的所有 BlockManager 进行管理的 BlockManagerMaster 及各个节点上对外提供 Block 上传与下载服务的 BlockTransferService。

6.7.1 BlockManager 的初始化

根据 5.7 节的介绍，每个 Driver 或 Executor 在创建自身的 SparkEnv 时都会创建 BlockManager。BlockManager 只有在其 initialize 方法被调用后才能发挥作用，其实现如代码清单 6-59 所示。

代码清单6-59 初始化BlockManager

```
def initialize(appId: String): Unit = {
  blockTransferService.init(this)
  shuffleClient.init(appId)
  blockReplicationPolicy = { // 设置Block的复制策略
    val priorityClass = conf.get(
      "spark.storage.replication.policy", classOf[RandomBlockReplicationPolicy].getName)
    val clazz = Utils.classForName(priorityClass)
    val ret = clazz.newInstance.asInstanceOf[BlockReplicationPolicy]
    logInfo(s"Using $priorityClass for block replication policy")
    ret
```

```
    }

    val id =
      BlockManagerId(executorId, blockTransferService.hostName, blockTransfer-
          Service.port, None)

    val idFromMaster = master.registerBlockManager(
      id,
      maxMemory,
      slaveEndpoint)
    // 生成当前BlockManager的BlockManagerId
    blockManagerId = if (idFromMaster != null) idFromMaster else id
    // 生成shuffleServerId
    shuffleServerId = if (externalShuffleServiceEnabled) {
      logInfo(s"external shuffle service port = $externalShuffleServicePort")
      BlockManagerId(executorId, blockTransferService.hostName, externalShuffle
          ServicePort)
    } else {
      blockManagerId
    }

    if (externalShuffleServiceEnabled && !blockManagerId.isDriver) {
      registerWithExternalShuffleServer() // 注册外部的Shuffle服务
    }

    logInfo(s"Initialized BlockManager: $blockManagerId")
  }
```

根据代码清单 6-59，BlockManager 初始化的步骤如下。

1）初始化 BlockTransferService，即 5.7 节介绍的 NettyBlockTransferService。

2）初始化 Shuffle 客户端。有关 shuffleClient 的内容将在 6.9 节详细介绍。

3）设置 Block 的复制策略。可以通过 spark.storage.replication.policy 属性来设置 Block 的复制策略，默认为 RandomBlockReplicationPolicy。这里使用了工具类 Utils 的 classForName 方法加载 Class，其具体实现可以参阅附录 A。

4）生成当前 BlockManager 的 BlockManagerId。BlockManager 在本地创建的 BlockManagerId 实际只是在向 BlockManagerMaster 注册 BlockManager 时，给 BlockManagerMaster 提供参考，BlockManagerMaster 将会创建一个包含了拓扑信息的新 BlockManagerId 作为正式分配给 BlockManager 的身份标识。

5）生成 shuffleServerId。当启用了外部 Shuffle 服务时将新建一个 BlockManagerId 作为 shuffleServerId，否则是 BlockManager 自身的 BlockManagerId。

6）当启用了外部 Shuffle 服务，并且当前 BlockManager 所在节点不是 Driver 时，需要注册外部的 Shuffle 服务。

6.7.2 BlockManager 提供的方法

BlockManager 提供了很多方法，这些方法将为 Spark 的存储体系提供支持，所以有必

要将它们介绍一番。读者在初次阅读本书时不必了解代码实现，只需要知道 BlockManager 提供了哪些方法即可，在介绍具体的调用时，可以回来查阅其实现，帮助我们理解。

1. reregister

此方法用于向 BlockManagerMaster 重新注册 BlockManager，并向 BlockManagerMaster 报告所有的 Block 信息。reregister 的实现如代码清单 6-60 所示。

代码清单6-60 reregister的实现

```
def reregister(): Unit = {
  master.registerBlockManager(blockManagerId, maxMemory, slaveEndpoint)
  reportAllBlocks()
}
```

根据代码清单 6-60，reregister 的步骤如下。

1）调用 BlockManagerMaster 的 registerBlockManager 方法（见代码清单 6-90）向 BlockManagerMaster 注册 BlockManager。

2）调用 reportAllBlocks 方法报告所有的 Block 信息。

reportAllBlocks 的实现如下。

```
private def reportAllBlocks(): Unit = {
  logInfo(s"Reporting ${blockInfoManager.size} blocks to the master.")
  for ((blockId, info) <- blockInfoManager.entries) {
    val status = getCurrentBlockStatus(blockId, info)
    if (info.tellMaster && !tryToReportBlockStatus(blockId, status)) {
      logError(s"Failed to report $blockId to master; giving up.")
      return
    }
  }
}
```

可以看到，reportAllBlocks 方法遍历 BlockInfoManager 管理的所有 BlockId 与 BlockInfo 的映射关系，并进行如下操作。

1）调用 getCurrentBlockStatus 方法（见代码清单 6-61），获取 Block 的状态信息 BlockStatus。

2）如果需要将 Block 的 BlockStatus 汇报给 BlockManagerMaster，则调用 tryToReportBlockStatus 方法（见代码清单 6-62），向 BlockManagerMaster 汇报此 Block 的状态信息。

代码清单6-61 获取Block的状态信息

```
private def getCurrentBlockStatus(blockId: BlockId, info: BlockInfo): BlockStatus = {
  info.synchronized {
    info.level match {
      case null =>
        BlockStatus.empty
      case level =>
        val inMem = level.useMemory && memoryStore.contains(blockId)
        val onDisk = level.useDisk && diskStore.contains(blockId)
        val deserialized = if (inMem) level.deserialized else false
```

```
        val replication = if (inMem  || onDisk) level.replication else 1
        val storageLevel = StorageLevel(
          useDisk = onDisk,
          useMemory = inMem,
          useOffHeap = level.useOffHeap,
          deserialized = deserialized,
          replication = replication)
        val memSize = if (inMem) memoryStore.getSize(blockId) else 0L
        val diskSize = if (onDisk) diskStore.getSize(blockId) else 0L
        BlockStatus(storageLevel, memSize, diskSize)
    }
  }
}
```

代码清单6-62 向BlockManagerMaster汇报Block的状态信息

```
private def tryToReportBlockStatus(
    blockId: BlockId,
    status: BlockStatus,
    droppedMemorySize: Long = 0L): Boolean = {
  val storageLevel = status.storageLevel
  val inMemSize = Math.max(status.memSize, droppedMemorySize)
  val onDiskSize = status.diskSize
  master.updateBlockInfo(blockManagerId, blockId, storageLevel, inMemSize, onDiskSize)
}
```

根据代码清单 6-62 可以看到，向 BlockManagerMaster 汇报 Block 的状态信息是通过调用 BlockManagerMaster 的 updateBlockInfo 方法完成的。BlockManagerMaster 的 updateBlockInfo 方法将向 BlockManagerMasterEndpoint 发送 UpdateBlockInfo 消息。BlockManager Master Endpoint 对 UpdateBlockInfo 消息的处理如代码清单 6-91 所示。

2. getLocalBytes

此方法用于从存储体系获取 BlockId 所对应 Block 的数据，并封装为 ChunkedByte-Buffer 后返回，其实现如代码清单 6-63 所示。

代码清单6-63 getLocalBytes的实现

```
def getLocalBytes(blockId: BlockId): Option[ChunkedByteBuffer] = {
  logDebug(s"Getting local block $blockId as bytes")
  if (blockId.isShuffle) {
    val shuffleBlockResolver = shuffleManager.shuffleBlockResolver
    Option(new ChunkedByteBuffer(
      shuffleBlockResolver.getBlockData(blockId.asInstanceOf[ShuffleBlockId]).
          nioByteBuffer()))
  } else {
    blockInfoManager.lockForReading(blockId).map { info => doGetLocalBytes
        (blockId, info) }
  }
}
```

根据代码清单 6-63，getLocalBytes 的执行步骤如下。

1）如果当前 Block 是 ShuffleBlock，那么调用 ShuffleManager 的 ShuffleBlockResolver 组件的 getBlockData 方法（ShuffleBlockResolver 将在 8.6 节详细介绍）获取 Block 数据，并封装为 ChunkedByteBuffer 返回。

2）如果当前 Block 不是 ShuffleBlock，那么首先获取 Block 的读锁，然后调用 doGetLocalBytes 方法获取 Block 数据。

doGetLocalBytes 的实现如代码清单 6-64 所示。

代码清单6-64 doGetLocalBytes的实现

```
private def doGetLocalBytes(blockId: BlockId, info: BlockInfo): ChunkedByteBuffer = {
  val level = info.level // 获取Block的存储级别
  logDebug(s"Level for block $blockId is $level")
  if (level.deserialized) { // Block没有被序列化，按照DiskStore、MemoryStore的顺序获
      取Block数据
    if (level.useDisk && diskStore.contains(blockId)) {
      diskStore.getBytes(blockId)
    } else if (level.useMemory && memoryStore.contains(blockId)) {
      serializerManager.dataSerializeWithExplicitClassTag(
        blockId, memoryStore.getValues(blockId).get, info.classTag)
    } else {
      handleLocalReadFailure(blockId)
    }
  } else { // Block被序列化了，那么按照MemoryStore、DiskStore的顺序获取Block数据
    if (level.useMemory && memoryStore.contains(blockId)) {
      memoryStore.getBytes(blockId).get
    } else if (level.useDisk && diskStore.contains(blockId)) {
      val diskBytes = diskStore.getBytes(blockId)
      maybeCacheDiskBytesInMemory(info, blockId, level, diskBytes).getOrElse
          (diskBytes)
    } else {
      handleLocalReadFailure(blockId)
    }
  }
}
```

根据代码清单 6-64，doGetLocalBytes 的执行步骤如下。

1）获取 Block 的存储级别。

2）如果 Block 的存储级别说明 Block 没有被序列化，那么按照 DiskStore、MemoryStore 的顺序，获取 Block 数据。

3）如果 Block 存储级别说明 Block 被序列化了，那么按照 MemoryStore、DiskStore 的顺序，获取 Block 数据。

3. getBlockData

此方法用于获取本地 Block 的数据，其实现如代码清单 6-65 所示。

代码清单6-65 获取本地Block的数据

```
override def getBlockData(blockId: BlockId): ManagedBuffer = {
  if (blockId.isShuffle) { // 当前Block是ShuffleBlock
```

```
      shuffleManager.shuffleBlockResolver.getBlockData(blockId.asInstanceOf
        [ShuffleBlockId])
    } else { // 当前Block不是ShuffleBlock
      getLocalBytes(blockId) match {
        case Some(buffer) => new BlockManagerManagedBuffer(blockInfoManager,
          blockId, buffer)
        case None =>
          reportBlockStatus(blockId, BlockStatus.empty)
          throw new BlockNotFoundException(blockId.toString)
      }
    }
  }
```

根据代码清单 6-65，getBlockData 的执行步骤如下。

1）如果当前 Block 是 ShuffleBlock，那么调用 ShuffleManager 的 ShuffleBlockResolver 组件的 getBlockData 方法获取 Block 数据。

2）如果当前 Block 不是 ShuffleBlock，那么调用 getLocalBytes 获取 Block 数据。如果调用 getLocalBytes 能够获取到 Block 数据，则封装为 BlockManagerManagedBuffer，否则调用 reportBlockStatus 方法（见代码清单 6-66）告诉 BlockManagerMaster，此 Block 不存在。

代码清单6-66 reportBlockStatus的实现

```
private def reportBlockStatus(
    blockId: BlockId,
    status: BlockStatus,
    droppedMemorySize: Long = 0L): Unit = {
  val needReregister = !tryToReportBlockStatus(blockId, status, droppedMemorySize)
  if (needReregister) {
    logInfo(s"Got told to re-register updating block $blockId")
    asyncReregister()
  }
  logDebug(s"Told master about block $blockId")
}
```

根据代码清单 6-66，reportBlockStatus 的执行步骤如下。

1）调用 tryToReportBlockStatus 方法向 BlockManagerMaster 汇报 BlockStatus。

2）如果第 1）步返回 true，则说明需要重新向 BlockManagerMaster 注册当前 BlockManager，因而调用 asyncReregister 方法向 BlockManagerMaster 异步注册 BlockManager。

asyncReregister 方法实际另起线程调用 reregister，来实现异步注册 BlockManager。asyncReregister 的实现如下。

```
private def asyncReregister(): Unit = {
  asyncReregisterLock.synchronized {
    if (asyncReregisterTask == null) {
      asyncReregisterTask = Future[Unit] {
        reregister()
        asyncReregisterLock.synchronized {
          asyncReregisterTask = null
```

```
        }
      }(futureExecutionContext)
    }
  }
}
```

4. putBytes

要介绍 putBytes，需要首先介绍 doPut。doPut 用于执行 Block 的写入，其实现如代码清单 6-67 所示。

代码清单6-67 doPut的实现

```
private def doPut[T](
    blockId: BlockId,
    level: StorageLevel,
    classTag: ClassTag[_],
    tellMaster: Boolean,
    keepReadLock: Boolean)(putBody: BlockInfo => Option[T]): Option[T] = {
  require(blockId != null, "BlockId is null")
  require(level != null && level.isValid, "StorageLevel is null or invalid")

  val putBlockInfo = {
    val newInfo = new BlockInfo(level, classTag, tellMaster)
    if (blockInfoManager.lockNewBlockForWriting(blockId, newInfo)) { // 获取Block
        的写锁
      newInfo
    } else {
      logWarning(s"Block $blockId already exists on this machine; not re-adding it")
      if (!keepReadLock) {
        // lockNewBlockForWriting returned a read lock on the existing block, so
            we must free it:
        releaseLock(blockId)
      }
      return None
    }
  }
  val startTimeMs = System.currentTimeMillis
  var exceptionWasThrown: Boolean = true
  val result: Option[T] = try {
    val res = putBody(putBlockInfo) // 执行Block写入
    exceptionWasThrown = false
    if (res.isEmpty) {
      // Block成功存储，执行锁降级或释放锁
      if (keepReadLock) {
        blockInfoManager.downgradeLock(blockId)
      } else {
        blockInfoManager.unlock(blockId)
      }
    } else { // Block存储失败，移除此Block
      removeBlockInternal(blockId, tellMaster = false)
      logWarning(s"Putting block $blockId failed")
    }
    res
  } finally {
```

```
      if (exceptionWasThrown) {
        logWarning(s"Putting block $blockId failed due to an exception")
        removeBlockInternal(blockId, tellMaster = tellMaster)
        addUpdatedBlockStatusToTaskMetrics(blockId, BlockStatus.empty)
      }
    }
    if (level.replication > 1) {
      logDebug("Putting block %s with replication took %s"
        .format(blockId, Utils.getUsedTimeMs(startTimeMs)))
    } else {
      logDebug("Putting block %s without replication took %s"
        .format(blockId, Utils.getUsedTimeMs(startTimeMs)))
    }
    result
  }
```

根据代码清单 6-67，我们看到 doPut 有一个函数参数 putBody，putBody 将执行真正的 Block 数据写入。doPut 的执行步骤如下。

1）获取 Block 的写锁。如果 Block 已经存在且不需要持有读锁，则需要当前线程释放持有的读锁。

2）调用 putBody，执行写入。

3）如果写入成功，则在需要保持读锁的情况下将写锁降级为读锁，在不需要保持读锁的情况下，释放所有锁。

4）如果写入失败，则调用 removeBlockInternal 方法移除此 Block。

5）如果写入时发生异常，也需要调用 removeBlockInternal 方法移除此 Block。此外，还需要调用 addUpdatedBlockStatusToTaskMetrics 方法更新任务度量信息。

在 doPut 方法中调用了 removeBlockInternal 方法来移除 Block，其实现如代码清单 6-68 所示。

代码清单6-68 removeBlockInternal的实现

```
  private def removeBlockInternal(blockId: BlockId, tellMaster: Boolean): Unit = {
    val removedFromMemory = memoryStore.remove(blockId)
    val removedFromDisk = diskStore.remove(blockId)
    if (!removedFromMemory && !removedFromDisk) {
      logWarning(s"Block $blockId could not be removed as it was not found on disk
          or in memory")
    }
    blockInfoManager.removeBlock(blockId)
    if (tellMaster) {
      reportBlockStatus(blockId, BlockStatus.empty)
    }
  }
```

根据代码清单 6-68，removeBlockInternal 的执行步骤如下。

1）从 MemoryStore 中移除 Block。

2）从 DiskStore 中移除 Block。

3）从 BlockInfoManager 中移除 Block 对应的 BlockInfo。

4）如果需要向 BlockManagerMaster 汇报 Block 状态，则调用 reportBlockStatus 方法（见代码清单 6-66）。

了解了 doPut，现在来看看 putBytes 的实现。putBytes 用于写入 Block 数据，其实现如代码清单 6-69 所示。

代码清单6-69 putBytes的实现

```
def putBytes[T: ClassTag](
    blockId: BlockId,
    bytes: ChunkedByteBuffer,
    level: StorageLevel,
    tellMaster: Boolean = true,
    encrypt: Boolean = false): Boolean = {
  require(bytes != null, "Bytes is null")

  val bytesToStore =
    if (encrypt && securityManager.ioEncryptionKey.isDefined) {
      try {
        val data = bytes.toByteBuffer
        val in = new ByteBufferInputStream(data, true)
        val byteBufOut = new ByteBufferOutputStream(data.remaining())
        val out = CryptoStreamUtils.createCryptoOutputStream(byteBufOut, conf,
          securityManager.ioEncryptionKey.get)
        try {
          ByteStreams.copy(in, out)
        } finally {
          in.close()
          out.close()
        }
        new ChunkedByteBuffer(byteBufOut.toByteBuffer)
      } finally {
        bytes.dispose()
      }
    } else {
      bytes
    }
  doPutBytes(blockId, bytesToStore, level, implicitly[ClassTag[T]], tellMaster)
}
```

根据代码清单 6-69，putBytes 首先对 Block 数据进行加密处理，然后调用 doPutBytes 方法写入 Block 数据。doPutBytes 的实现如下。

```
private def doPutBytes[T](
    blockId: BlockId,
    bytes: ChunkedByteBuffer,
    level: StorageLevel,
    classTag: ClassTag[T],
    tellMaster: Boolean = true,
    keepReadLock: Boolean = false): Boolean = {
  doPut(blockId, level, classTag, tellMaster = tellMaster, keepReadLock =
      keepReadLock) { info =>
```

```
val startTimeMs = System.currentTimeMillis
val replicationFuture = if (level.replication > 1) {
  Future { // 创建异步线程通过调用replicate方法复制Block数据到其他节点的存储体系中
    replicate(blockId, bytes, level, classTag)
  }(futureExecutionContext)
} else {
  null
}
val size = bytes.size
if (level.useMemory) { // 优先写入内存
  val putSucceeded = if (level.deserialized) {
    val values =
      serializerManager.dataDeserializeStream(blockId, bytes.toInputStream())
          (classTag)
    memoryStore.putIteratorAsValues(blockId, values, classTag) match {
      case Right(_) => true
      case Left(iter) =>
        iter.close()
        false
    }
  } else {
    memoryStore.putBytes(blockId, size, level.memoryMode, () => bytes)
  }
  if (!putSucceeded && level.useDisk) { // 内存不足，写入磁盘
    logWarning(s"Persisting block $blockId to disk instead.")
    diskStore.putBytes(blockId, bytes)
  }
} else if (level.useDisk) { // 不能使用内存时，写入磁盘
  diskStore.putBytes(blockId, bytes)
}
val putBlockStatus = getCurrentBlockStatus(blockId, info)
val blockWasSuccessfullyStored = putBlockStatus.storageLevel.isValid
if (blockWasSuccessfullyStored) {
  info.size = size
  if (tellMaster && info.tellMaster) {
    reportBlockStatus(blockId, putBlockStatus) // 向BlockManagerMaster报告
        Block的状态
  }
  addUpdatedBlockStatusToTaskMetrics(blockId, putBlockStatus)
}
logDebug("Put block %s locally took %s".format(blockId, Utils.getUsedTimeMs
    (startTimeMs)))
if (level.replication > 1) {
  // 等待异步的复制线程完成
  try {
    Await.ready(replicationFuture, Duration.Inf)
  } catch {
    case NonFatal(t) =>
      throw new Exception("Error occurred while waiting for replication to
          finish", t)
  }
}
if (blockWasSuccessfullyStored) {
  None
} else {
  Some(bytes)
```

```
    }
  }.isEmpty
}
```

根据 doPutBytes 的实现，其首先定义了偏函数，这个偏函数将作为 doPut 的 putBody 参数，然后调用 doPut 方法，doPut 方法将调用此偏函数，偏函数写入数据的步骤如下。

1）如果 Block 的 StorageLevel 的复制数量大于 1，则创建异步线程通过调用 replicate 方法复制 Block 数据到其他节点的存储体系中。

2）如果 Block 的 StorageLevel 允许数据写入内存，首先写入内存。如果内存不足且 Block 的 StorageLevel 允许数据写入磁盘，则写入磁盘。

3）如果 Block 的 StorageLevel 允许数据写入磁盘，则写入磁盘。

4）调用 getCurrentBlockStatus 方法（见代码清单 6-61）获取当前 Block 的状态。如果此状态说明 Block 数据成功存储到存储体系，那么调用 reportBlockStatus 方法（见代码清单 6-66）向 BlockManagerMaster 报告 Block 的状态，还调用 addUpdatedBlockStatusToTaskMetrics 方法更新任务度量信息。

5. putBlockData

此方法用于将 Block 数据写入本地，其实现如代码清单 6-70 所示。

代码清单6-70 将Block数据写入本地

```
override def putBlockData(
    blockId: BlockId,
    data: ManagedBuffer,
    level: StorageLevel,
    classTag: ClassTag[_]): Boolean = {
  putBytes(blockId, new ChunkedByteBuffer(data.nioByteBuffer()), level)(classTag)
}
```

根据代码清单 6-70，putBlockData 实际调用了 putBytes。

6. getStatus

此方法用于获取 Block 的状态，其实现如代码清单 6-71 所示。

代码清单6-71 获取Block的状态

```
def getStatus(blockId: BlockId): Option[BlockStatus] = {
  blockInfoManager.get(blockId).map { info =>
    val memSize = if (memoryStore.contains(blockId)) memoryStore.getSize(blockId) else 0L
    val diskSize = if (diskStore.contains(blockId)) diskStore.getSize(blockId) else 0L
    BlockStatus(info.level, memSize = memSize, diskSize = diskSize)
  }
}
```

7. getMatchingBlockIds

此方法用于获取匹配过滤器条件的 BlockId 的序列，其实现如代码清单 6-72 所示。

代码清单6-72 getMatchingBlockIds的实现

```
def getMatchingBlockIds(filter: BlockId => Boolean): Seq[BlockId] = {
  (blockInfoManager.entries.map(_._1) ++ diskBlockManager.getAllBlocks())
    .filter(filter)
    .toArray
    .toSeq
}
```

根据代码清单6-72，我们看到除了从BlockInfoManager的entries缓存中获取BlockId外，还需要从DiskBlockManager中获取。这是因为DiskBlockManager中可能存在BlockInfoManager不知道的Block。

8. getLocalValues

此方法用于从本地的BlockManager中获取Block数据，其实现如代码清单6-73所示。

代码清单6-73 从本地的BlockManager中获取Block数据

```
def getLocalValues(blockId: BlockId): Option[BlockResult] = {
  logDebug(s"Getting local block $blockId")
  blockInfoManager.lockForReading(blockId) match { // 获取BlockId所对应的读锁
    case None =>
      logDebug(s"Block $blockId was not found")
      None
    case Some(info) =>
      val level = info.level
      logDebug(s"Level for block $blockId is $level")
      if (level.useMemory && memoryStore.contains(blockId)) {
      //优先从MemoryStore中读取Block数据
        val iter: Iterator[Any] = if (level.deserialized) {
          memoryStore.getValues(blockId).get
        } else {
          serializerManager.dataDeserializeStream(
            blockId, memoryStore.getBytes(blockId).get.toInputStream())(info.classTag)
        }
        val ci = CompletionIterator[Any, Iterator[Any]](iter, releaseLock(blockId))
        Some(new BlockResult(ci, DataReadMethod.Memory, info.size))
      } else if (level.useDisk && diskStore.contains(blockId)) {
      // 从DiskStore中读取Block数据
        val iterToReturn: Iterator[Any] = {
          val diskBytes = diskStore.getBytes(blockId)
          if (level.deserialized) {
            val diskValues = serializerManager.dataDeserializeStream(
              blockId,
              diskBytes.toInputStream(dispose = true))(info.classTag)
            maybeCacheDiskValuesInMemory(info, blockId, level, diskValues)
          } else {
            val stream = maybeCacheDiskBytesInMemory(info, blockId, level,
               diskBytes)
              .map {_.toInputStream(dispose = false)}
              .getOrElse { diskBytes.toInputStream(dispose = true) }
            serializerManager.dataDeserializeStream(blockId, stream)(info.classTag)
          }
        }
```

```
            val ci = CompletionIterator[Any, Iterator[Any]](iterToReturn, releaseLock(blockId))
            Some(new BlockResult(ci, DataReadMethod.Disk, info.size))
          } else {
            handleLocalReadFailure(blockId)
          }
      }
    }
```

根据代码清单 6-73，getLocalValues 的执行步骤如下。

1）获取 BlockId 所对应的读锁。

2）优先从 MemoryStore 中读取 Block 数据。

3）从 DiskStore 中读取 Block 数据。

9. getRemoteBytes

getRemoteBytes 方法的作用为从远端的 BlockManager 以序列化的字节形式获取 Block 数据。但在此之前，首先介绍获取 Block 位置信息的方法 getLocations，其实现如代码清单 6-74 所示。

代码清单6-74 getLocations的实现

```
private def getLocations(blockId: BlockId): Seq[BlockManagerId] = {
  val locs = Random.shuffle(master.getLocations(blockId))
  val (preferredLocs, otherLocs) = locs.partition { loc => blockManagerId.host ==
     loc.host }
  preferredLocs ++ otherLocs
}
```

根据代码清单 6-74，getLocations 方法的执行步骤如下。

1）调用 BlockManagerMaster 的 getLocations 方法获取所需 Block 所在的所有位置信息（即 BlockManagerId）序列，并随机打乱。

2）将 BlockManagerId 序列划分为 preferredLocs 与 otherLocs。preferredLocs 中的 BlockManagerId 所标识的 BlockManager 与当前 BlockManager 位于同一机器上，而 otherLocs 中的 BlockManagerId 所标识的 BlockManager 与当前 BlockManager 位于不同机器上。

3）将 preferredLocs 中的 BlockManagerId 放置在 otherLocs 中的 BlockManagerId 前面，构成一个新的序列返回。这一步骤涉及 Block 的本地性选择。

有了对 getLocations 方法的了解，现在来看看 getRemoteBytes 的实现，如代码清单 6-75 所示。

代码清单6-75 getRemoteBytes的实现

```
def getRemoteBytes(blockId: BlockId): Option[ChunkedByteBuffer] = {
  logDebug(s"Getting remote block $blockId")
  require(blockId != null, "BlockId is null")
  var runningFailureCount = 0
  var totalFailureCount = 0
  val locations = getLocations(blockId) // 获取Block所在的所有位置信息
```

```
    val maxFetchFailures = locations.size
    var locationIterator = locations.iterator
    while (locationIterator.hasNext) {
      val loc = locationIterator.next()
      logDebug(s"Getting remote block $blockId from $loc")
      val data = try { // 以同步方式从远端下载Block
        blockTransferService.fetchBlockSync(
          loc.host, loc.port, loc.executorId, blockId.toString).nioByteBuffer()
      } catch {
        case NonFatal(e) =>
          runningFailureCount += 1
          totalFailureCount += 1
          if (totalFailureCount >= maxFetchFailures) { // 已经做了最大努力，但还是没能
              下载成功
            logWarning(s"Failed to fetch block after $totalFailureCount fetch failures. " +
              s"Most recent failure cause:", e)
            return None
          }
          // 刷新Block所在的所有位置信息
          if (runningFailureCount >= maxFailuresBeforeLocationRefresh) {
            locationIterator = getLocations(blockId).iterator
            logDebug(s"Refreshed locations from the driver " +
              s"after ${runningFailureCount} fetch failures.")
            runningFailureCount = 0
          }
          null
      }
      if (data != null) {
        return Some(new ChunkedByteBuffer(data))
      }
      logDebug(s"The value of block $blockId is null")
    }
    logDebug(s"Block $blockId not found")
    None
  }
```

根据代码清单 6-75，getRemoteBytes 的执行步骤如下。

1）调用 getLocations 方法获取 Block 所在的所有位置信息序列 locations。

2）设置 maxFetchFailures 等于 locations 的大小（即最大获取失败次数）。

3）从 locations 序列中按顺序取出下一个 BlockManagerId，并调用 BlockTransferService 的 fetchBlockSync 方法（见代码清单 6-109），以同步方式从远端下载 Block。

4）如果调用 fetchBlockSync 方法时发生了异常，则增加下载失败次数（runningFailureCount）和下载失败总数（totalFailureCount）。当 totalFailureCount 大于等于 maxFetchFailures 时，说明已经作了最大努力。当 runningFailureCount 大于等于 maxFailuresBeforeLocationRefresh 时，则会重新调用 getLocations 方法刷新 Block 所在的所有位置信息，并将 runningFailureCount 清零。

5）如果第 3）步获取到数据，那么将得到的数据封装为 ChunkedByteBuffer 并返回，否则回到第 3）步继续执行。

6）如果没有获取到数据，则返回 None。

10. get

此方法用于优先从本地获取 Block 数据，当本地获取不到所需的 Block 数据，再从远端获取 Block 数据，其实现如代码清单 6-76 所示。

代码清单6-76 获取Block数据

```
def get[T: ClassTag](blockId: BlockId): Option[BlockResult] = {
  val local = getLocalValues(blockId)
  if (local.isDefined) {
    logInfo(s"Found block $blockId locally")
    return local
  }
  val remote = getRemoteValues[T](blockId)
  if (remote.isDefined) {
    logInfo(s"Found block $blockId remotely")
    return remote
  }
  None
}
```

11. downgradeLock

此方法用于将当前线程持有的 Block 的写锁降级为读锁，其实现如代码清单 6-77 所示。

代码清单6-77 锁降级

```
def downgradeLock(blockId: BlockId): Unit = {
  blockInfoManager.downgradeLock(blockId)
}
```

downgradeLock 实际代理了 BlockInfoManager 的 downgradeLock 方法。

12. releaseLock

此方法用于当前线程对持有的 Block 的锁进行释放，其实现如代码清单 6-78 所示。

代码清单6-78 释放锁

```
def releaseLock(blockId: BlockId): Unit = {
  blockInfoManager.unlock(blockId)
}
```

releaseLock 实际调用了 BlockInfoManager 的 unlock 方法。

13. registerTask

此方法用于将任务尝试线程注册到 BlockInfoManager，其实现如代码清单 6-79 所示。

代码清单6-79 注册任务

```
def registerTask(taskAttemptId: Long): Unit = {
  blockInfoManager.registerTask(taskAttemptId)
}
```

registerTask 实际代理了 BlockInfoManager 的 registerTask 方法。

14. releaseAllLocksForTask

此方法用于任务尝试线程对持有的所有 Block 的锁进行释放，其实现如代码清单 6-80 所示。

代码清单6-80 releaseAllLocksForTask的实现

```
def releaseAllLocksForTask(taskAttemptId: Long): Seq[BlockId] = {
  blockInfoManager.releaseAllLocksForTask(taskAttemptId)
}
```

releaseAllLocksForTask 实际代理了 BlockInfoManager 的 releaseAllLocksForTask 方法。

15. getOrElseUpdate

getOrElseUpdate 方法用于获取 Block。如果 Block 存在，则获取此 Block 并返回 BlockResult，否则调用 makeIterator 方法计算 Block，并持久化后返回 BlockResult 或 Iterator。getOrElseUpdate 的实现如代码清单 6-81 所示。

代码清单6-81 getorElsevpdate的实现

```
def getOrElseUpdate[T](
      blockId: BlockId,
      level: StorageLevel,
      classTag: ClassTag[T],
      makeIterator: () => Iterator[T]): Either[BlockResult, Iterator[T]] = {
    get[T](blockId)(classTag) match { // 从本地或远端的BlockManager获取Block
      case Some(block) =>
        return Left(block)
      case _ =>
      // Need to compute the block.
  }
    doPutIterator(blockId, makeIterator, level, classTag, keepReadLock = true)
        match {
      case None => // Block已经成功存储到内存
        val blockResult = getLocalValues(blockId).getOrElse {
        releaseLock(blockId)
        throw new SparkException(s"get() failed for block $blockId even though we
            held a lock")
      }
      releaseLock(blockId)
      Left(blockResult)
    case Some(iter) => // Block存储到内存时发生了错误
     Right(iter)
  }
}
```

根据代码清单 6-81，getOrElseUpdate 的执行步骤如下。

1）从本地或远端的 BlockManager 获取 Block。如果能够获取到 Block，则返回 Left。

2）调用 doPutIterator 方法计算、持久化 Block。doPutIterator 方法的实现与 doPutBytes

十分相似，都定义了计算、持久化 Block 的偏函数，并以此偏函数作为 putBody 参数调用 doPut，所以不再赘述。

3）doPutIterator 方法的返回结果为 None，说明计算得到的 Block 已经成功存储到内存，因此再次读取此 Block。

4）doPutIterator 方法的返回结果匹配 Some，说明计算得到的 Block 存储到内存时发生了错误。

16. putIterator

此方法用于将 Block 数据写入存储体系，其实现如代码清单 6-82 所示。

代码清单6-82 putIterator的实现

```
def putIterator[T: ClassTag](
    blockId: BlockId,
    values: Iterator[T],
    level: StorageLevel,
    tellMaster: Boolean = true): Boolean = {
  require(values != null, "Values is null")
  doPutIterator(blockId, () => values, level, implicitly[ClassTag[T]], tellMaster)
      match {
    case None =>
      true
    case Some(iter) =>
      iter.close()
      false
  }
}
```

根据代码清单 6-82，putIterator 内部实际也调用了 doPutIterator 方法。当 doPutIterator 返回 None，说明计算得到的 Block 已经成功存储到内存，因此再次读取此 Block。doPutIterator 方法的返回结果匹配 Some，则说明计算得到的 Block 存储到内存时发生了错误。

17. getDiskWriter

此方法用于创建并获取 DiskBlockObjectWriter，通过 DiskBlockObjectWriter 可以跳过对 DiskStore 的使用，直接将数据写入磁盘。getDiskWriter 的实现如代码清单 6-83 所示。

代码清单6-83 getDiskWriter的实现

```
def getDiskWriter(
    blockId: BlockId,
    file: File,
    serializerInstance: SerializerInstance,
    bufferSize: Int,
    writeMetrics: ShuffleWriteMetrics): DiskBlockObjectWriter = {
  val syncWrites = conf.getBoolean("spark.shuffle.sync", false)
  new DiskBlockObjectWriter(file, serializerManager, serializerInstance, bufferSize,
    syncWrites, writeMetrics, blockId)
}
```

根据代码清单 6-83 我们知道，属性 spark.shuffle.sync 将决定 DiskBlockObjectWriter 把数据写入磁盘时是采用同步方式还是异步方式，默认是异步方式。

18. getSingle

此方法的作用为获取由单个对象组成的 Block，其实现如代码清单 6-84 所示。

代码清单6-84 getSingle的实现

```
def getSingle[T: ClassTag](blockId: BlockId): Option[T] = {
  get[T](blockId).map(_.data.next().asInstanceOf[T])
}
```

根据代码清单 6-84，我们知道 getSingle 其实调用了 get 方法。

19. putSingle

此方法用于将由单个对象组成的 Block 写入存储体系，其实现如代码清单 6-85 所示。

代码清单6-85 putSingle的实现

```
def putSingle[T: ClassTag](
    blockId: BlockId,
    value: T,
    level: StorageLevel,
    tellMaster: Boolean = true): Boolean = {
  putIterator(blockId, Iterator(value), level, tellMaster)
}
```

根据代码清单 6-85，putSingle 实际调用了 putIterator。

20. dropFromMemory

此方法用于从内存中删除 Block，当 Block 的存储级别允许写入磁盘，Block 将被写入磁盘。此方法主要在内存不足，需要从内存腾出空闲空间时使用。dropFromMemory 的实现如代码清单 6-86 所示。

代码清单6-86 dropFromMemory的实现

```
private[storage] override def dropFromMemory[T: ClassTag](
    blockId: BlockId,
    data: () => Either[Array[T], ChunkedByteBuffer]): StorageLevel = {
  logInfo(s"Dropping block $blockId from memory")
  // 确认当前任务尝试线程是否已经持有BlockId对应的写锁
  val info = blockInfoManager.assertBlockIsLockedForWriting(blockId)
  var blockIsUpdated = false
  val level = info.level
  // 将Block写入磁盘
  if (level.useDisk && !diskStore.contains(blockId)) {
    logInfo(s"Writing block $blockId to disk")
    data() match {
      case Left(elements) =>
        diskStore.put(blockId) { fileOutputStream =>
          serializerManager.dataSerializeStream(
            blockId,
```

```
            fileOutputStream,
            elements.toIterator)(info.classTag.asInstanceOf[ClassTag[T]])
        }
      case Right(bytes) =>
        diskStore.putBytes(blockId, bytes)
    }
    blockIsUpdated = true
  }
  // 将内存中的Block删除
  val droppedMemorySize =
    if (memoryStore.contains(blockId)) memoryStore.getSize(blockId) else 0L
  val blockIsRemoved = memoryStore.remove(blockId)
  if (blockIsRemoved) {
    blockIsUpdated = true
  } else {
    logWarning(s"Block $blockId could not be dropped from memory as it does not
        exist")
  }
  val status = getCurrentBlockStatus(blockId, info)
  if (info.tellMaster) {
    reportBlockStatus(blockId, status, droppedMemorySize) // 向BlockManagerMaster
        报告Block状态
  }
  if (blockIsUpdated) {
    addUpdatedBlockStatusToTaskMetrics(blockId, status) // 更新任务度量信息
  }
  status.storageLevel // 返回Block的存储级别
}
```

根据代码清单 6-86，dropFromMemory 的执行步骤如下。

1）确认当前任务尝试线程是否已经持有 BlockId 对应的写锁。

2）如果 Block 对应的存储级别允许 Block 使用磁盘，并且 Block 尚未写入磁盘，则调用 DiskStore 的 put 方法或 putBytes 方法将 Block 写入磁盘。

3）如果 MemoryStore 中存在 Block，则调用 MemoryStore 的 getSize 方法获取将要从内存中删除的 Block 的大小 droppedMemorySize。

4）调用 MemoryStore 的 remove 方法将内存中的 Block 删除。

5）调用 getCurrentBlockStatus 方法（见代码清单 6-61）获取 Block 的当前状态。

6）如果 BlockInfo 的 tellMaster 属性为 true，则调用 reportBlockStatus 方法（见代码清单 6-66）向 BlockManagerMaster 报告 Block 状态。

7）当 Block 写入了磁盘或 Block 从内存中删除，则调用 addUpdatedBlockStatusToTaskMetrics 方法更新任务度量信息。

8）返回 Block 的存储级别。

21. removeBlock

此方法的作用为删除指定的 Block，其实现如代码清单 6-87 所示。

代码清单6-87 removeBlock的实现

```
def removeBlock(blockId: BlockId, tellMaster: Boolean = true): Unit = {
  logDebug(s"Removing block $blockId")
  blockInfoManager.lockForWriting(blockId) match { // 获取指定Block的写锁
    case None =>
      // The block has already been removed; do nothing.
      logWarning(s"Asked to remove block $blockId, which does not exist")
    case Some(info) => // 从存储体系中删除Block
      removeBlockInternal(blockId, tellMaster = tellMaster && info.tellMaster)
      addUpdatedBlockStatusToTaskMetrics(blockId, BlockStatus.empty)
  }
}
```

根据代码清单 6-87，removeBlock 方法的执行步骤如下。

1）获取指定 Block 的写锁。

2）如果获取到写锁，则调用 removeBlockInternal 方法（见代码清单 6-68）从存储体系中删除 Block。

22. removeRdd

此方法的作用为移除属于指定 RDD 的所有 Block，其实现如代码清单 6-88 所示。

代码清单6-88 removeRdd的实现

```
def removeRdd(rddId: Int): Int = {
  logInfo(s"Removing RDD $rddId")
  val blocksToRemove = blockInfoManager.entries.flatMap(_._1.asRDDId).filter(_.
     rddId == rddId)
  blocksToRemove.foreach { blockId => removeBlock(blockId, tellMaster = false) }
  blocksToRemove.size
}
```

根据代码清单 6-88，removeRdd 的执行步骤如下。

1）从 BlockInfoManager 的 entries 中找出所有的 RDDBlockId，并过滤出其 rddId 属性等于指定 rddId 的所有 RDDBlockId。

2）调用 removeBlock 方法删除过滤出来的所有 RDDBlockId。

23. removeBroadcast

此方法的作用为移除属于指定 Broadcast 的所有 Block，其实现如代码清单 6-89 所示。

代码清单6-89 removeBroadcast的实现

```
def removeBroadcast(broadcastId: Long, tellMaster: Boolean): Int = {
  logDebug(s"Removing broadcast $broadcastId")
  val blocksToRemove = blockInfoManager.entries.map(_._1).collect {
    case bid @ BroadcastBlockId(`broadcastId`, _) => bid
  }
  blocksToRemove.foreach { blockId => removeBlock(blockId, tellMaster) }
  blocksToRemove.size
}
```

根据代码清单 6-89，可以看出 removeBroadcast 与 removeRdd 的实现相似。

6.8 BlockManagerMaster 对 BlockManager 的管理

BlockManagerMaster 的作用是对存在于 Executor 或 Driver 上的 BlockManager 进行统一管理。Executor 与 Driver 关于 BlockManager 的交互都依赖于 BlockManagerMaster，比如 Executor 需要向 Driver 发送注册 BlockManager、更新 Executor 上 Block 的最新信息、询问所需要 Block 目前所在的位置及当 Executor 运行结束需要将此 Executor 移除等。但是 Driver 与 Executor 却位于不同机器中，该怎么实现呢？

根据 5.7 节的内容，我们知道 Driver 上的 BlockManagerMaster 会实例化并且注册 BlockManagerMasterEndpoint。无论是 Driver 还是 Executor，它们的 BlockManagerMaster 的 driverEndpoint 属性都将持有 BlockManagerMasterEndpoint 的 RpcEndpointRef。无论是 Driver 还是 Executor，每个 BlockManager 都拥有自己的 BlockManagerSlaveEndpoint，且 BlockManager 的 slaveEndpoint 属性保存着各自 BlockManagerSlaveEndpoint 的 RpcEndpointRef。BlockManagerMaster 负责发送消息，BlockManagerMasterEndpoint 负责消息的接收与处理，BlockManagerSlaveEndpoint 则接收 BlockManagerMasterEndpoint 下发的命令。图 6-1 中也展现了这几者之间的关系。

6.8.1 BlockManagerMaster 的职责

BlockManagerMaster 负责发送各种与存储体系相关的消息，这些消息的类型如下。

- RemoveExecutor（移除 Executor）。
- RegisterBlockManager（注册 BlockManager）。
- UpdateBlockInfo（更新 Block 信息）。
- GetLocations（获取 Block 的位置）。
- GetLocationsMultipleBlockIds（获取多个 Block 的位置）。
- GetPeers（获取其他 BlockManager 的 BlockManagerId）。
- GetExecutorEndpointRef（获取 Executor 的 EndpointRef 引用）。
- RemoveBlock（移除 Block）。
- RemoveRdd（移除 Rdd Block）。
- RemoveShuffle（移除 Shuffle Block）。
- RemoveBroadcast（移除 Broadcast Block）。
- GetMemoryStatus（获取指定的 BlockManager 的内存状态）。
- GetStorageStatus（获取存储状态）。
- GetBlockStatus（获取 Block 的状态）。
- GetMatchingBlockIds（获取匹配过滤条件的 Block）。

- HasCachedBlocks（指定的 Executor 上是否有缓存的 Block）。
- StopBlockManagerMaster（停止 BlockManagerMaster）。

可以看到，BlockManagerMaster 能够发送的消息类型多种多样，为了更容易理解，本节以 RegisterBlockManager 为例，来介绍 BlockManagerMaster 发挥的作用。BlockManagerMaster 提供的 registerBlockManager 方法负责发送 RegisterBlockManager 消息，其实现如代码清单 6-90 所示。

代码清单6-90 注册BlockManager

```
def registerBlockManager(
    blockManagerId: BlockManagerId,
    maxMemSize: Long,
    slaveEndpoint: RpcEndpointRef): BlockManagerId = {
  logInfo(s"Registering BlockManager $blockManagerId")
  val updatedId = driverEndpoint.askWithRetry[BlockManagerId](
    RegisterBlockManager(blockManagerId, maxMemSize, slaveEndpoint))
  logInfo(s"Registered BlockManager $updatedId")
  updatedId
}
```

根据代码清单 6-90，注册 BlockManager 的实质是向 BlockManagerMasterEndpoint 发送 RegisterBlockManager 消息，RegisterBlockManager 将携带要注册的 BlockManager 的 blockManagerId、最大内存大小及 slaveEndpoint（即 BlockManagerSlaveEndpoint）。

BlockManagerMaster 对于其他类型的消息也提供了对应的方法实现，实现过程都与 registerBlockManager 方法类似，留给感兴趣的读者自行阅读。

6.8.2 BlockManagerMasterEndpoint 详解

BlockManagerMasterEndpoint 接收 Driver 或 Executor 上 BlockManagerMaster 发送的消息，对所有的 BlockManager 统一管理。BlockManagerMasterEndpoint 定义了一些管理 BlockManager 的属性，这些属性的作用如下。

- blockManagerInfo：BlockManagerId 与 BlockManagerInfo 之间映射关系的缓存。
- blockManagerIdByExecutor：Executor ID 与 BlockManagerId 之间映射关系的缓存。
- blockLocations：BlockId 与存储了此 BlockId 对应 Block 的 BlockManager 的 BlockManagerId 之间的一对多关系缓存。
- topologyMapper：对集群所有节点的拓扑结构的映射。

BlockManagerMasterEndpoint 重写了特质 RpcEndpoint 的 receiveAndReply 方法，用于接收 BlockManager 相关的消息，其实现如代码清单 6-91 所示。

代码清单6-91 receiveAndReply的实现

```
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
  case RegisterBlockManager(blockManagerId, maxMemSize, slaveEndpoint) =>
    context.reply(register(blockManagerId, maxMemSize, slaveEndpoint))
```

```
    case _updateBlockInfo @
        UpdateBlockInfo(blockManagerId, blockId, storageLevel, deserializedSize,
            size) =>
      context.reply(updateBlockInfo(blockManagerId, blockId, storageLevel,
          deserializedSize, size))
      listenerBus.post(SparkListenerBlockUpdated(BlockUpdatedInfo(_updateBlockInfo)))
    case GetLocations(blockId) =>
      context.reply(getLocations(blockId))
    case GetLocationsMultipleBlockIds(blockIds) =>
      context.reply(getLocationsMultipleBlockIds(blockIds))
    case GetPeers(blockManagerId) =>
      context.reply(getPeers(blockManagerId))
    case GetExecutorEndpointRef(executorId) =>
      context.reply(getExecutorEndpointRef(executorId))
    case GetMemoryStatus =>
      context.reply(memoryStatus)
    case GetStorageStatus =>
      context.reply(storageStatus)
    case GetBlockStatus(blockId, askSlaves) =>
      context.reply(blockStatus(blockId, askSlaves))
    case GetMatchingBlockIds(filter, askSlaves) =>
      context.reply(getMatchingBlockIds(filter, askSlaves))
    case RemoveRdd(rddId) =>
      context.reply(removeRdd(rddId))
    case RemoveShuffle(shuffleId) =>
      context.reply(removeShuffle(shuffleId))
    case RemoveBroadcast(broadcastId, removeFromDriver) =>
      context.reply(removeBroadcast(broadcastId, removeFromDriver))
    case RemoveBlock(blockId) =>
      removeBlockFromWorkers(blockId)
      context.reply(true)
    case RemoveExecutor(execId) =>
      removeExecutor(execId)
      context.reply(true)
    case StopBlockManagerMaster =>
      context.reply(true)
      stop()
    case BlockManagerHeartbeat(blockManagerId) =>
      context.reply(heartbeatReceived(blockManagerId))
    case HasCachedBlocks(executorId) =>
      blockManagerIdByExecutor.get(executorId) match {
        case Some(bm) =>
          if (blockManagerInfo.contains(bm)) {
            val bmInfo = blockManagerInfo(bm)
            context.reply(bmInfo.cachedBlocks.nonEmpty)
          } else {
            context.reply(false)
          }
        case None => context.reply(false)
      }
  }
```

根据代码清单6-91，BlockManagerMasterEndpoint接收的消息类型正好与BlockManagerMaster所发送的消息一一对应。本节依然选择RegisterBlockManager消息，来介绍

BlockManagerMasterEndpoint 是如何接收和处理 RegisterBlockManager 消息的。根据代码清单 6-91，BlockManagerMasterEndpoint 在接收到 RegisterBlockManager 消息后，将调用 register 方法。register 的实现如代码清单 6-92 所示。

代码清单6-92 register的实现

```
private def register(
    idWithoutTopologyInfo: BlockManagerId,
    maxMemSize: Long,
    slaveEndpoint: RpcEndpointRef): BlockManagerId = {
  val id = BlockManagerId( // 生成BlockManagerId
    idWithoutTopologyInfo.executorId,
    idWithoutTopologyInfo.host,
    idWithoutTopologyInfo.port,
    topologyMapper.getTopologyForHost(idWithoutTopologyInfo.host))

  val time = System.currentTimeMillis()
  if (!blockManagerInfo.contains(id)) {
    blockManagerIdByExecutor.get(id.executorId) match {
      case Some(oldId) =>
        logError("Got two different block manager registrations on same executor - "
            + s" will replace old one $oldId with new one $id")
        removeExecutor(id.executorId) //移除blockManagerIdByExecutor中的缓存信息
      case None =>
    }
    logInfo("Registering block manager %s with %s RAM, %s".format(
      id.hostPort, Utils.bytesToString(maxMemSize), id))

    blockManagerIdByExecutor(id.executorId) = id

    blockManagerInfo(id) = new BlockManagerInfo(
      id, System.currentTimeMillis(), maxMemSize, slaveEndpoint)
  }
  listenerBus.post(SparkListenerBlockManagerAdded(time, id, maxMemSize))
  id
}
```

根据代码清单 6-92，register 方法的执行步骤如下。

1）生成 BlockManagerId。BlockManagerId 由 executorId（Executor 的标识）、host（块传输服务的 host）、port（块传输服务的 port）及拓扑信息组成。

2）如果 blockManagerInfo 缓存中不存在此 BlockManagerId，则进入下一步，否则进入第 6）步。

3）移除 blockManagerIdByExecutor、blockManagerInfo、blockLocations 等缓存中与此 BlockManagerId 有关的所有缓存信息。removeExecutor 方法的实现如代码清单 6-93 所示。其中调用的 removeBlockManager 方法的实现很简单，感兴趣的读者可以自己阅读其实现。

代码清单6-93 removeExecutor的实现

```
private def removeExecutor(execId: String) {
  logInfo("Trying to remove executor " + execId + " from BlockManagerMaster.")
```

```
    blockManagerIdByExecutor.get(execId).foreach(removeBlockManager)
  }
```

4）将 executorId 与新创建的 BlockManagerId 的对应关系添加到 blockManagerIdByExecutor 中。

5）将 BlockManagerId 与 BlockManagerInfo 的对应关系添加到缓存 blockManagerInfo。

6）向 listenerBus 投递 SparkListenerBlockManagerAdded 类型的事件。根据 3.3 节的内容，listenerThread 线程最终将触发对所有 SparkListener 的 onBlockManagerAdded 方法的调用，进而达到监控的目的。

本节以注册 BlockManager 为例，介绍了 BlockManagerMasterEndpoint 的作用。其他类型的消息处理都是类似的，故此不再赘述。

6.8.3 BlockManagerSlaveEndpoint 详解

BlockManagerSlaveEndpoint 用于接收 BlockManagerMasterEndpoint 的命令并执行相应的操作。BlockManagerSlaveEndpoint 也重写了 RpcEndpoint 的 receiveAndReply 方法，其实现如代码清单 6-94 所示。

代码清单6-94 BlockManagerSlaveEndpoint的receiveAndReply实现

```
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
  case RemoveBlock(blockId) =>
    doAsync[Boolean]("removing block " + blockId, context) {
      blockManager.removeBlock(blockId)
      true
    }
  case RemoveRdd(rddId) =>
    doAsync[Int]("removing RDD " + rddId, context) {
      blockManager.removeRdd(rddId)
    }
  case RemoveShuffle(shuffleId) =>
    doAsync[Boolean]("removing shuffle " + shuffleId, context) {
      if (mapOutputTracker != null) {
        mapOutputTracker.unregisterShuffle(shuffleId)
      }
      SparkEnv.get.shuffleManager.unregisterShuffle(shuffleId)
    }
  case RemoveBroadcast(broadcastId, _) =>
    doAsync[Int]("removing broadcast " + broadcastId, context) {
      blockManager.removeBroadcast(broadcastId, tellMaster = true)
    }
  case GetBlockStatus(blockId, _) =>
    context.reply(blockManager.getStatus(blockId))
  case GetMatchingBlockIds(filter, _) =>
    context.reply(blockManager.getMatchingBlockIds(filter))
  case TriggerThreadDump =>
    context.reply(Utils.getThreadDump())
}
```

书籍是人类进步的阶梯

本节以 Driver 删除 Block 为例，来介绍 BlockManagerSlaveEndpoint 的作用。Driver 节点删除 Block 依赖于 BlockManagerMaster 提供的 removeBlock 方法，其实现如代码清单 6-95 所示。

代码清单6-95 removeBlock的实现

```
def removeBlock(blockId: BlockId) {
  driverEndpoint.askWithRetry[Boolean](RemoveBlock(blockId))
}
```

根据代码清单 6-95，BlockManagerMaster 提供的 removeBlock 方法实际向 BlockManagerMasterEndpoint 发送 RemoveBlock 类型的消息。根据代码清单 6-91 中 BlockManagerMasterEndpoint 的 receiveAndReply 方法的实现，BlockManagerMasterEndpoint 首先调用 removeBlockFromWorkers，然后向客户端回复 true。removeBlockFromWorkers 方法的实现如代码清单 6-96 所示。

代码清单6-96 removeBlockFromWorkers的实现

```
private def removeBlockFromWorkers(blockId: BlockId) {
  val locations = blockLocations.get(blockId)
  if (locations != null) {
    locations.foreach { blockManagerId: BlockManagerId =>
      val blockManager = blockManagerInfo.get(blockManagerId)
      if (blockManager.isDefined) {
        blockManager.get.slaveEndpoint.ask[Boolean](RemoveBlock(blockId))
      }
    }
  }
}
```

根据代码清单 6-96，removeBlockFromWorkers 的执行步骤如下。

1）从缓存 blockLocations 中获取 BlockId 所对应 Block 实际存储的位置集合 locations。

2）遍历 locations，向每个节点的 BlockManagerSlaveEndpoint 发送 RemoveBlock 消息。

根据代码清单 6-94，BlockManagerSlaveEndpoint 接收到 RemoveBlock 消息后，将调用 BlockManager 的 removeBlock 删除 Block。

6.9 Block 传输服务

根据之前的介绍，我们知道 BlockTransferService 是 BlockManager 的子组件之一。抽象类 BlockTransferService 有两个实现：用于测试的 MockBlockTransferService 及 5.7 节曾经介绍的 NettyBlockTransferService。根据 5.7 节的介绍，我们知道 BlockManager 实际采用了 NettyBlockTransferService 提供的 Block 传输服务。

读者可能奇怪为什么要把由 Netty 实现的网络服务组件也放到存储体系里面。由于 Spark 是分布式部署的，每个 Task（准确说是任务尝试）最终都运行在不同的机器节点上。map 任务的输出结果直接存储到 map 任务所在机器的存储体系中，reduce 任务极有可能不

书籍是人类进步的阶梯

在同一机器上运行，所以需要远程下载 map 任务的中间输出。NettyBlockTransferService 提供了可以被其他节点的客户端访问的 Shuffle 服务。

有了 Shuffle 的服务端，那么也需要相应的 Shuffle 客户端，以便当前节点将 Block 上传到其他节点或者从其他节点下载 Block 到本地。BlockManager 中创建 Shuffle 客户端的代码如下。

```
private[spark] val shuffleClient = if (externalShuffleServiceEnabled) {
  val transConf = SparkTransportConf.fromSparkConf(conf, "shuffle", numUsableCores)
  new ExternalShuffleClient(transConf, securityManager,
    securityManager.isAuthenticationEnabled(),
    securityManager.isSaslEncryptionEnabled())
} else {
  blockTransferService
}
```

从上面的代码可以知道，如果部署了外部的 Shuffle 服务，则需要配置 spark.shuffle.service.enabled 属性为 true（此属性将决定 externalShuffleServiceEnabled 的值，默认是 false），此时将创建 ExternalShuffleClient。但在默认情况下，NettyBlockTransferService 也会作为 Shuffle 的客户端。

小贴士：熟悉 Hadoop YARN 的用户可能已经发现 Spark 与 Hadoop 一样，都采用 Netty 来实现 Shuffle Server。

6.9.1 初始化 NettyBlockTransferService

NettyBlockTransferService 只有在其 init 方法被调用，即被初始化后才提供服务。根据 6.7.1 节的内容，BlockManager 在初始化的时候，将调用 NettyBlockTransferService 的 init 方法。NettyBlockTransferService 的 init 方法的实现如代码清单 6-97 所示。

代码清单6-97 初始化NettyBlockTransferService

```
override def init(blockDataManager: BlockDataManager): Unit = {
   val rpcHandler = new NettyBlockRpcServer(conf.getAppId, serializer,
blockDataManager)
  var serverBootstrap: Option[TransportServerBootstrap] = None
  var clientBootstrap: Option[TransportClientBootstrap] = None
  if (authEnabled) {
    serverBootstrap = Some(new SaslServerBootstrap(transportConf, securityManager))
    clientBootstrap = Some(new SaslClientBootstrap(transportConf, conf.getAppId,
        securityManager,
      securityManager.isSaslEncryptionEnabled()))
  }
  transportContext = new TransportContext(transportConf, rpcHandler)
  clientFactory = transportContext.createClientFactory(clientBootstrap.toSeq.asJava)
  server = createServer(serverBootstrap.toList)
  appId = conf.getAppId
  logInfo(s"Server created on ${hostName}:${server.getPort}")
}
```

书籍是人类进步的阶梯

根据代码清单 6-97，NettyBlockTransferService 的初始化步骤如下。

1）创建 NettyBlockRpcServer。NettyBlockRpcServer 继承了我们熟悉的 RpcHandler，根据 3.2 节的内容，我们知道服务端对客户端的 Block 读写请求的处理都交给了 RpcHandler 的实现类，因此 NettyBlockRpcServer 将处理 Block 块的 RPC 请求。

2）准备客户端引导程序 TransportClientBootstrap 和服务端引导程序 TransportServerBootstrap。

3）创建 TransportContext。

4）创建传输客户端工厂 TransportClientFactory。

5）创建 TransportServer。

6）获取当前应用的 ID。

由于 NettyBlockTransferService 初始化的过程，与 5.3 节 RPC 环境的创建过程非常相似，所以不再赘述。

6.9.2 NettyBlockRpcServer 详解

由于 RPC 框架的内容已在 3.2 节有详细介绍，所以我们重点来看看 NettyBlockTransferService 与 NettyRpcEnv 的最大区别——使用的 RpcHandler 的实现类不同，NettyRpcEnv 采用了 NettyRpcHandler，而 NettyBlockTransferService 则采用了 NettyBlockRpcServer。

1. OneForOneStreamManager 的实现

NettyBlockRpcServer 中使用了 OneForOneStreamManager 来提供一对一的流服务。OneForOneStreamManager 实现了 StreamManager 的 registerChannel、getChunk、connectionTerminated、checkAuthorization、registerStream 五个方法，所以根据 3.2.6 节对 TransportRequestHandler 的介绍，我们知道 TransportRequestHandler 的 processFetchRequest 方法用到了 StreamManager 的 checkAuthorization、registerChannel 和 getChunk 三个方法，OneForOneStreamManager 将处理 ChunkFetchRequest 类型的消息。

OneForOneStreamManager 中使用 StreamState 来维护流的状态，所以我们需要先认识 StreamState，StreamState 的实现如代码清单 6-98 所示。

代码清单6-98 StreamState的实现

```
private static class StreamState {
  final String appId;
  final Iterator<ManagedBuffer> buffers;

  Channel associatedChannel = null;
  int curChunk = 0;
  StreamState(String appId, Iterator<ManagedBuffer> buffers) {
    this.appId = appId;
    this.buffers = Preconditions.checkNotNull(buffers);
  }
}
```

书籍是人类进步的阶梯

根据代码清单 6-98，StreamState 包含以下属性。

1）appId：请求流所属的应用程序 ID。此属性只有在 ExternalShuffleClient 启用后才会用到。

2）buffers：ManagedBuffer 的缓冲。

3）associatedChannel：与当前流相关联的 Channel。

4）curChunk：为了保证客户端按顺序每次请求一个块，所以用此属性跟踪客户端当前接收到的 ManagedBuffer 的索引。

有了对 StreamState 的了解，我们来看看 OneForOneStreamManager 的成员属性。

5）nextStreamId：用于生成数据流的标识，类型为 AtomicLong。

6）streams：维护 streamId 与 StreamState 之间映射关系的缓存。

认识了以上介绍的属性，现在来看看 OneForOneStreamManager 中实现的方法。

7）checkAuthorization 方法（见代码清单 6-99）用以校验客户端是否有权限从给定的流中读取。

代码清单6-99 checkAuthorization的实现

```
@Override
public void checkAuthorization(TransportClient client, long streamId) {
  if (client.getClientId() != null) {
    StreamState state = streams.get(streamId);
    Preconditions.checkArgument(state != null, "Unknown stream ID.");
    if (!client.getClientId().equals(state.appId)) {
      throw new SecurityException(String.format(
        "Client %s not authorized to read stream %d (app %s).",
        client.getClientId(),
        streamId,
        state.appId));
    }
  }
}
```

如果没有配置对管道进行 SASL 认证（参见 3.2.6 节代码清单 3-35 服务端的 SaslServerBootstrap 对管道进行 SASL 认证的内容），TransportClient 的 clientId 为 null，因而实际上并不走权限检查。当启用了 SASL 认证，客户端需要给 TransportClient 的 clientId 赋值，因此才会走此检查。checkAuthorization 方法检查的办法很简单，即将 TransportClient 的 clientId 属性值与 streamId 对应的 StreamState 的 appId 的值进行相等比较。

8）registerChannel 方法（见代码清单 6-100）用于注册管道，其实际的作用是将一个流和一条（只能是一条）客户端的 TCP 连接关联起来，这可以保证对于单个的流只会有一个客户端读取。流关闭之后就永远不能够重用了。

代码清单6-100 registerChannel的实现

```
@Override
public void registerChannel(Channel channel, long streamId) {
```

书籍是人类进步的阶梯

```
    if (streams.containsKey(streamId)) {
      streams.get(streamId).associatedChannel = channel;
    }
  }
```

9）getChunk 方法（见代码清单 6-101）用于获取单个的块（块被封装为 ManagedBuffer）。

代码清单6-101　getChunk的实现

```
@Override
public ManagedBuffer getChunk(long streamId, int chunkIndex) {
  StreamState state = streams.get(streamId); // 从streams中获取StreamState
  if (chunkIndex != state.curChunk) {
    throw new IllegalStateException(String.format(
      "Received out-of-order chunk index %s (expected %s)", chunkIndex, state.
          curChunk));
  } else if (!state.buffers.hasNext()) {
    throw new IllegalStateException(String.format(
      "Requested chunk index beyond end %s", chunkIndex));
  }
  state.curChunk += 1; // 将StreamState的curChunk加1，为下次接收请求做好准备
  ManagedBuffer nextChunk = state.buffers.next(); // 从buffers缓冲中获取ManagedBuffer
  if (!state.buffers.hasNext()) { // buffers缓冲中的ManagedBuffer，已经全部被客户端获取
    logger.trace("Removing stream id {}", streamId);
    streams.remove(streamId);
  }
  return nextChunk;
}
```

根据代码清单 6-101，getChunk 的执行步骤如下。

① 从 streams 中获取 StreamState。如果要获取的块的索引与 StreamState 的 curChunk 属性不相等，则说明顺序有问题。如果要获取的块的索引超出了 buffers 缓冲的大小，这说明请求了一个超出范围的块。

② 将 StreamState 的 curChunk 加 1，为下次接收请求做好准备。

③ 从 buffers 缓冲中获取 ManagedBuffer。如果 buffers 缓冲已经迭代到了末端，那么说明当前流的块已经全部被客户端获取了，需要将 streamId 与对应的 StreamState 从 streams 中移除。

④ 返回获取的 ManagedBuffer。

10）registerStream 方法（见代码清单 6-102）用于向 OneForOneStreamManager 的 streams 缓存中注册流。

代码清单6-102　registerStream的实现

```
public long registerStream(String appId, Iterator<ManagedBuffer> buffers) {
  long myStreamId = nextStreamId.getAndIncrement();
  streams.put(myStreamId, new StreamState(appId, buffers));
  return myStreamId;
}
```

书籍是人类进步的阶梯

根据代码清单6-102，registerStream方法首先生成一个新的streamId，然后创建StreamState对象，最后将streamId与StreamState对象之间的映射关系放入streams中。

2. NettyBlockRpcServer的实现

介绍了NettyBlockRpcServer的内部组件OneForOneStreamManager，下面来看看NettyBlockRpcServer的实现，如代码清单6-103所示。

代码清单6-103 NettyBlockRpcServer的实现

```
class NettyBlockRpcServer(
    appId: String,
    serializer: Serializer,
    blockManager: BlockDataManager)
  extends RpcHandler with Logging {

  private val streamManager = new OneForOneStreamManager()

  override def receive(
      client: TransportClient,
      rpcMessage: ByteBuffer,
      responseContext: RpcResponseCallback): Unit = {
  val message = BlockTransferMessage.Decoder.fromByteBuffer(rpcMessage)
  logTrace(s"Received request: $message")

  message match {
    case openBlocks: OpenBlocks => // 打开（读取）Block
      val blocks: Seq[ManagedBuffer] =
        openBlocks.blockIds.map(BlockId.apply).map(blockManager.getBlockData)
      val streamId = streamManager.registerStream(appId, blocks.iterator.asJava)
      logTrace(s"Registered streamId $streamId with ${blocks.size} buffers")
      responseContext.onSuccess(new StreamHandle(streamId, blocks.size).toByteBuffer)

    case uploadBlock: UploadBlock => // 上传Block
      val (level: StorageLevel, classTag: ClassTag[_]) = {
        serializer
          .newInstance()
          .deserialize(ByteBuffer.wrap(uploadBlock.metadata))
          .asInstanceOf[(StorageLevel, ClassTag[_])]
      }
      val data = new NioManagedBuffer(ByteBuffer.wrap(uploadBlock.blockData))
      val blockId = BlockId(uploadBlock.blockId)
      blockManager.putBlockData(blockId, data, level, classTag)
      responseContext.onSuccess(ByteBuffer.allocate(0))
  }
}

override def getStreamManager(): StreamManager = streamManager
}
```

根据代码清单6-103，NettyBlockRpcServer实现了需要回复客户端的receive和getStreamManager两个方法，其中getStreamManager方法将返回OneForOneStreamManager。NettyBlockRpcServer的receive方法将分别接收以下两种消息。

1）OpenBlocks：打开（读取）Block。NettyBlockRpcServer 对这种消息的处理步骤如下。

① 取出 OpenBlocks 消息携带的 BlockId 数组。

② 调用 BlockManager 的 getBlockData 方法，获取数组中每一个 BlockId 对应的 Block（返回值为 ManagedBuffer 的序列）。

③ 调用 OneForOneStreamManager 的 registerStream 方法，将 ManagedBuffer 序列注册到 OneForOneStreamManager 的 streams 缓存。

④ 创建 StreamHandle 消息（包含 streamId 和 ManagedBuffer 序列的大小），并通过响应上下文回复客户端。

2）UploadBlock：上传 Block。NettyBlockRpcServer 对这种消息的处理步骤如下。

① 对 UploadBlock 消息携带的元数据 metadata 进行反序列化，得到存储级别（StorageLevel）和类型标记（上传 Block 的类型）。

② 将 UploadBlock 消息携带的 Block 数据（即 blockData），封装为 NioManagedBuffer。

③ 获取 UploadBlock 消息携带的 BlockId。

④ 调用 BlockManager 的 putBlockData 方法，将 Block 存入本地存储体系。

⑤ 通过响应上下文回复客户端。

6.9.3 Shuffle 客户端

根据前面的介绍，如果没有部署外部的 Shuffle 服务，即 spark.shuffle.service.enabled 属性为 false 时，NettyBlockTransferService 不但通过 OneForOneStreamManager 与 NettyBlockRpcServer 对外提供 Block 上传与下载的服务，也将作为默认的 Shuffle 客户端。NettyBlockTransferService 作为 Shuffle 客户端，具有发起上传和下载请求并接收服务端响应的能力。NettyBlockTransferService 的两个方法——fetchBlocks 和 uploadBlock 将帮我们达到目的。

1. 发送下载远端 Block 的请求

NettyBlockTransferService 的 fetchBlocks 方法的实现如代码清单 6-104 所示。

代码清单6-104 fetchBlocks的实现

```
override def fetchBlocks(
    host: String,
    port: Int,
    execId: String,
    blockIds: Array[String],
    listener: BlockFetchingListener): Unit = {
  logTrace(s"Fetch blocks from $host:$port (executor id $execId)")
  try {
    val blockFetchStarter = new RetryingBlockFetcher.BlockFetchStarter {
      override def createAndStart(blockIds: Array[String], listener:
          BlockFetchingListener) {
        val client = clientFactory.createClient(host, port)
```

```
          new OneForOneBlockFetcher(client, appId, execId, blockIds.toArray, listener).start()
        }
      }

      val maxRetries = transportConf.maxIORetries()
      if (maxRetries > 0) { // 创建Block的重试线程
        new RetryingBlockFetcher(transportConf, blockFetchStarter, blockIds, listener).start()
      } else {
        blockFetchStarter.createAndStart(blockIds, listener)
      }
    } catch {
      case e: Exception =>
        logError("Exception while beginning fetchBlocks", e)
        blockIds.foreach(listener.onBlockFetchFailure(_, e))
    }
  }
```

根据代码清单 6-104，fetchBlocks 的执行步骤如下。

1）创建 RetryingBlockFetcher.BlockFetchStarter 的匿名实现类的实例 blockFetchStarter，此匿名类实现了 BlockFetchStarter 接口的 createAndStart 方法。

2）获取 spark.$module.io.maxRetries 属性（NettyBlockTransferService 的 module 为 shuffle）的值作为下载请求的最大重试次数 maxRetries。

3）只有配置的 spark.shuffle.io.maxRetries 属性大于 0，maxRetries 才有效，此时将创建 RetryingBlockFetcher 并调用 RetryingBlockFetcher 的 start 方法，否则直接调用 blockFetchStarter 的 createAndStart 方法。

RetryingBlockFetcher 的 start 方法只是调用了 fetchAllOutstanding 方法。

```
public void start() {
  fetchAllOutstanding();
}
```

fetchAllOutstanding 方法的实现如代码清单 6-105 所示。

代码清单6-105 fetchAllOutstanding的实现

```
private void fetchAllOutstanding() {
  // Start by retrieving our shared state within a synchronized block.
  String[] blockIdsToFetch;
  int numRetries;
  RetryingBlockFetchListener myListener;
  synchronized (this) {
    blockIdsToFetch = outstandingBlocksIds.toArray(new String[outstanding-
        BlocksIds.size()]);
    numRetries = retryCount;
    myListener = currentListener;
  }

  // 启动对Block的获取，并可能对获取失败的Block进行重试
  try {
    fetchStarter.createAndStart(blockIdsToFetch, myListener);
  } catch (Exception e) {
```

```
    logger.error(String.format("Exception while beginning fetch of %s outstanding
        blocks %s",
      blockIdsToFetch.length, numRetries > 0 ? "(after " + numRetries + "
          retries)" : ""), e);

    if (shouldRetry(e)) {
      initiateRetry();
    } else {
      for (String bid : blockIdsToFetch) {
        listener.onBlockFetchFailure(bid, e);
      }
    }
  }
}
```

根据代码清单 6-105，fetchAllOutstanding 方法的执行步骤如下。

1）调用 fetchStarter（即 blockFetchStarter）的 createAndStart 方法。其中 myListener 为 RetryingBlockFetchListener，RetryingBlockFetchListener 是 BlockFetchingListener 的实现类。

2）如果上一步执行时抛出了异常，则调用 shouldRetry 方法判断是否需要重试。shouldRetry 方法（见代码清单 6-106）的判断依据是：异常是 IOException 并且当前的重试次数 retryCount 小于最大重试次数 maxRetries。

代码清单6-106　判断是否需要重试

```
private synchronized boolean shouldRetry(Throwable e) {
  boolean isIOException = e instanceof IOException
    || (e.getCause() != null && e.getCause() instanceof IOException);
  boolean hasRemainingRetries = retryCount < maxRetries;
  return isIOException && hasRemainingRetries;
}
```

3）当需要重试时调用 initiateRetry 方法再次重试，如代码清单 6-107 所示。

代码清单6-107　重试的实现

```
private synchronized void initiateRetry() {
  retryCount += 1;
  currentListener = new RetryingBlockFetchListener();

  logger.info("Retrying fetch ({}/{}) for {} outstanding blocks after {} ms",
    retryCount, maxRetries, outstandingBlocksIds.size(), retryWaitTime);

  executorService.submit(new Runnable() {
    @Override
    public void run() {
      Uninterruptibles.sleepUninterruptibly(retryWaitTime, TimeUnit.MILLISECONDS);
      fetchAllOutstanding();
    }
  });
}
```

根据代码清单 6-107，重试的步骤如下。

① 重试次数 retryCount 加 1。

② 新建 RetryingBlockFetchListener。

③ 使用线程池 executorService（由 Executors.newCachedThreadPool 方法创建）提交新的任务，任务实际为调用 fetchAllOutstanding 方法。由此可以看出，下载远端 Block 的重试机制是异步的。

根据对发送获取远端 Block 的请求的分析，无论是请求一次还是异步多次重试，最后都落实到调用 blockFetchStarter 的 createAndStart 方法。blockFetchStarter 的 createAndStart 方法（见代码清单 6-104）首先创建 TransportClient，然后创建 OneForOneBlockFetcher 并调用其 start 方法。

客户端在需要下载远端节点的 Block 时，将创建 OneForOneBlockFetcher，然后使用 OneForOneBlockFetcher 完成 Block 的下载。了解 OneForOneBlockFetcher 之前，应该首先理解其中的各个成员属性。

- client：用于向服务端发送请求的 TransportClient。
- openMessage：即 OpenBlocks。OpenBlocks 将携带远端节点的 appId(应用程序标识)、execId（Executor 标识）和 blockIds（BlockId 的数组），这表示从远端的哪个实例获取哪些 Block，并且知道是哪个远端 Executor 生成的 Block。
- blockIds：BlockId 的数组。与 openMessage 的 blockIds 属性一致。
- listener：类型为 BlockFetchingListener，将在获取 Block 成功或失败时被回调。
- chunkCallback：获取块成功或失败时回调，配合 BlockFetchingListener 使用。
- streamHandle：根据代码清单 6-103，客户端给服务端发送 OpenBlocks 消息后，服务端会在 OneForOneStreamManager 的 streams 缓存中缓存从存储体系中读取到的 ManagedBuffer 序列，并生成与 ManagedBuffer 序列对应的 streamId，然后将 streamId 和 ManagedBuffer 序列的大小封装为 StreamHandle 消息返回给客户端，客户端的 streamHandle 属性将持有此 StreamHandle 消息。

有了对 OneForOneBlockFetcher 的各个属性的了解，现在来看看 OneForOneBlockFetcher 是如何获取远端 Block 的。OneForOneBlockFetcher 的 start 方法用于获取远端 Block，其实现如代码清单 6-108 所示。

代码清单6-108 获取远端Block

```
public void start() {
  if (blockIds.length == 0) {
    throw new IllegalArgumentException("Zero-sized blockIds array");
  }

  client.sendRpc(openMessage.toByteBuffer(), new RpcResponseCallback() {
    @Override
    public void onSuccess(ByteBuffer response) {
      try {
        streamHandle = (StreamHandle) BlockTransferMessage.Decoder.fromByteBuffer
```

```
            (response);
          logger.trace("Successfully opened blocks {}, preparing to fetch chunks.",
              streamHandle);

          for (int i = 0; i < streamHandle.numChunks; i++) { // 多次请求获取Block
            client.fetchChunk(streamHandle.streamId, i, chunkCallback);
          }
        } catch (Exception e) {
          logger.error("Failed while starting block fetches after success", e);
          failRemainingBlocks(blockIds, e);
        }
      }

      @Override
      public void onFailure(Throwable e) {
        logger.error("Failed while starting block fetches", e);
        failRemainingBlocks(blockIds, e);
      }
    });
  }
```

根据代码清单 6-108，OneForOneBlockFetcher 获取远端 Block 的步骤如下。

1）校验要获取的 Block 的数量。如果要获取的 BlockId 的数组的大小为零，则抛出 IllegalArgumentException 异常。

2）调用 TransportClient 的 sendRpc 方法发送 openMessage（即 OpenBlocks）消息，并向客户端的 outstandingRpcs 缓存注册匿名的 RpcResponseCallback 实现。

3）客户端等待接收服务端的响应（此时服务端正在进行代码清单 6-103 所介绍的处理）。

4）当客户端收到服务端的响应时，将从 outstandingRpcs 缓存取出代码清单 6-108 中的匿名 RpcResponseCallback 实现，以服务端的响应类别（RpcResponse 或 RpcFailure）分别回调匿名 RpcResponseCallback 的不同方法（onSuccess 或 onFailure）。

5）如果回调 onSuccess，则先将服务端的 response 反序列化为 StreamHandle 类型，然后根据 StreamHandle 的 numChunks 属性的大小，按照索引由低到高，遍历调用 TransportClient 的 fetchChunk 方法逐个获取 Block。

TransportClient 的 sendRpc 方法和 fetchChunk 方法已在 3.2.8 节详细介绍过，此处不再赘述。

客户端发送下载 Block 的 RPC 请求及服务端接收此请求的内容，目前都已经进行了分析和讲解，为了能让读者把客户端和服务端的内容串起来，这里以图 6-9 来展示整个 Block 的下载流程。

图 6-9 中有很多序号，它们所代表的处理逻辑如下。

序号①：客户端调用 ShuffleClient（默认为 NettyBlockTransferService）的 fetchBlocks 方法下载远端节点的 Block，由于未指定 spark.shuffle.io.maxRetries 属性，所以将直接调用 BlockFetchStarter 的 createAndStart 方法。

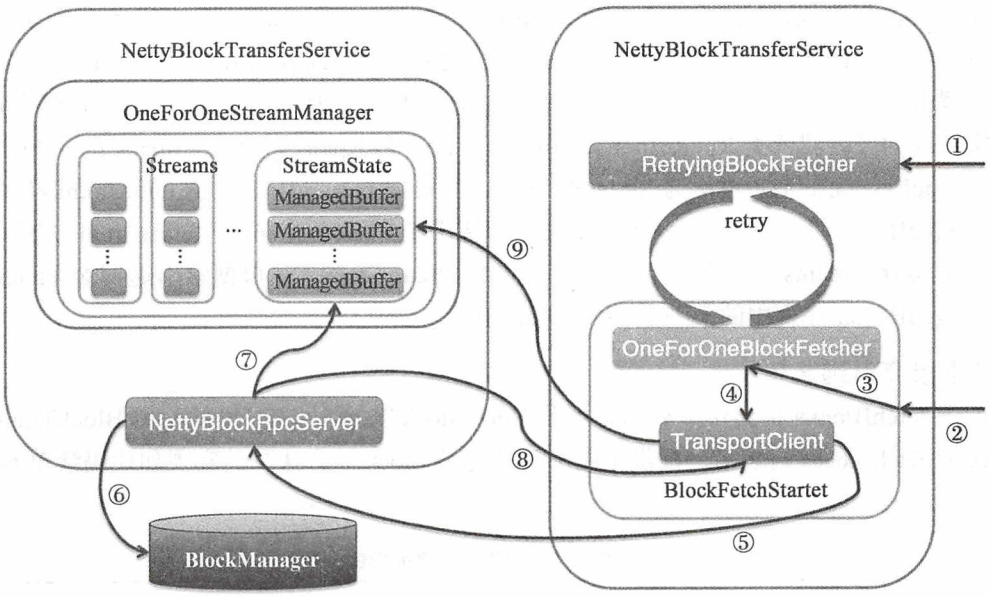


图 6-9 Shuffle 之 Block 的下载流程

序号②：客户端调用 ShuffleClient（默认为 NettyBlockTransferService）的 fetchBlocks 方法下载远端节点的 Block，由于未指定 park.shuffle.io.maxRetries 属性，所以首先调用 RetryingBlockFetcher 的 start 方法。RetryingBlockFetcher 内部实际也将调用 BlockFetch Starter 的 createAndStart 方法，并且在发生 IOException 且重试次数 retryCount 小于最大重试次数 maxRetries 时，多次尝试调用 BlockFetchStarter 的 createAndStart 方法。

序号③：BlockFetchStarter 的 createAndStart 方法实际将调用 OneForOneBlockFetcher 的 start 方法。

序号④：OneForOneBlockFetcher 将调用 TransportClient 的 sendRpc 方法发送 OpenBlocks 消息。OpenBlocks 携带着远端节点的 appId（应用程序标识）、execId（Executor 标识）和 blockIds（BlockId 的数组）等信息。此外，TransportClient 还将向 outstandingRpcs 中注册本次请求对应的匿名回调类 RpcResponseCallback。

序号⑤：NettyBlockRpcServer 接收到客户端发来的 OpenBlocks 消息，并获取携带的 BlockId 的数组信息。

序号⑥：NettyBlockRpcServer 对 BlockId 数组中的每个 BlockId，通过调用 BlockManager 的 getBlockData 方法获取数据块的信息（ManagedBuffer），所有读取的 Block 数据将构成一个 ManagedBuffer 类型的序列。

序号⑦：NettyBlockRpcServer 调用 OneForOneStreamManager 的 registerStream 方法生成 streamId 和 StreamState，并将 streamId 与 StreamState 的映射关系缓存在 streams 中。StreamState 中将包含着 appId（应用程序标识）和 ManagedBuffer 序列。

序号⑧：创建 StreamHandle（包含着 streamId 和 ManagedBuffer 序列的大小），并通过回调方法将 StreamHandle 封装为 RpcResponse，最后向客户端发送 RpcResponse。

序号⑨：TransportClient 接收到 RpcResponse 消息后，从 outstandingRpcs 中查找到本次请求对应的匿名回调类 RpcResponseCallback，此匿名 RpcResponseCallback 的 onSuccess 方法将 RpcResponse 消息解码为 StreamHandle，并根据 StreamHandle 的 numChunks（块数，即 ManagedBuffer 序列的大小），按照索引逐个调用 TransportClient 的 fetchChunk 方法从远端节点的缓存 streams 中找到与 streamId 对应的 StreamState，并根据索引返回 StreamState 的 ManagedBuffer 序列中的某一个 ManagedBuffer。

2. 同步下载远端 Block

由于 fetchBlocks 是异步下载的，因此 NettyBlockTransferService 的父类 BlockTransferService 对 fetchBlocks 封装后，提供了同步下载远端 Block 的方法，其实现如代码清单 6-109 所示。

代码清单6-109 同步下载远端Block

```
def fetchBlockSync(host: String, port: Int, execId: String, blockId: String): ManagedBuffer = {
  val result = Promise[ManagedBuffer]()
  fetchBlocks(host, port, execId, Array(blockId),
    new BlockFetchingListener {
      override def onBlockFetchFailure(blockId: String, exception: Throwable):
          Unit = {
        result.failure(exception)
      }
      override def onBlockFetchSuccess(blockId: String, data: ManagedBuffer):
          Unit = {
        val ret = ByteBuffer.allocate(data.size.toInt)
        ret.put(data.nioByteBuffer())
        ret.flip()
        result.success(new NioManagedBuffer(ret))
      }
    })
  ThreadUtils.awaitResult(result.future, Duration.Inf)
}
```

根据代码清单 6-109，其实质为调用 fetchBlocks。

3. 发送向远端上传 Block 的请求

NettyBlockTransferService 的 uploadBlock 方法的实现如代码清单 6-110 所示。

代码清单6-110 uploadBlock的实现

```
override def uploadBlock(
    hostname: String,
    port: Int,
    execId: String,
    blockId: BlockId,
    blockData: ManagedBuffer,
    level: StorageLevel,
```

```
    classTag: ClassTag[_]): Future[Unit] = {
  val result = Promise[Unit]()
  val client = clientFactory.createClient(hostname, port)
  val metadata = JavaUtils.bufferToArray(serializer.newInstance().serialize
    ((level, classTag)))
  val array = JavaUtils.bufferToArray(blockData.nioByteBuffer())
  client.sendRpc(new UploadBlock(appId, execId, blockId.toString, metadata,
    array).toByteBuffer,
    new RpcResponseCallback {
      override def onSuccess(response: ByteBuffer): Unit = {
        logTrace(s"Successfully uploaded block $blockId")
        result.success((): Unit)
      }
      override def onFailure(e: Throwable): Unit = {
        logError(s"Error while uploading block $blockId", e)
        result.failure(e)
      }
    })
  result.future
}
```

根据代码清单 6-110，uploadBlock 方法的执行步骤如下。

1）创建一个空 Promise，调用方将持有此 Promise 的 Future。

2）创建 TransportClient。

3）将存储级别 StorageLevel 和类型标记 classTag 等元数据序列化。

4）调用 ManagedBuffer（实际是实现类 NettyManagedBuffer）的 nioByteBuffer 方法将 Block 的数据转换或者复制为 Nio 的 ByteBuffer 类型。

5）调用 TransportClient 的 sendRpc 方法发送 RPC 消息 UploadBlock。UploadBlock 消息的 appId 是应用程序的标识，execId 是上传目的地的 Executor 的标识，RpcResponseCallback 是匿名的实现类。上传成功则回调匿名 RpcResponseCallback 的 onSuccess 方法，进而调用 Promise 的 success 方法；上传失败则回调匿名 RpcResponseCallback 的 onFailure 方法，进而调用 Promise 的 failure 方法。

客户端发送上传 Block 的 RPC 请求及服务端接收此请求的内容，目前都已经进行了分析和讲解，为了能让读者把客户端和服务端的内容串起来，这里以图 6-10 来展示整个 Block 的上传流程。

图 6-10 中有很多序号，它们所代表的处理逻辑如下。

序号①：客户端从本地的 BlockManager 中读取 BlockId 对应的 Block，并转换为 ManagedBuffer 类型。

序号②：客户端调用 ShuffleClient（默认为 NettyBlockTransferService）的 uploadBlock 方法上传 Block 到远端节点。NettyBlockTransferService 将 Block、Block 的存储级别及类型标记等信息进行序列化，生成 metadata 和 array。

序号③：NettyBlockTransferService 调用 TransportClient 的 sendRpc 方法发送 UploadBlock 消息。UploadBlock 携带着目标节点的 appId（应用程序标识），execId（Executor 标识），

BlockId 及 metadata 和 array 等信息。此外，TransportClient 还将向 outstandingRpcs 中注册本次请求对应的匿名回调类 RpcResponseCallback。

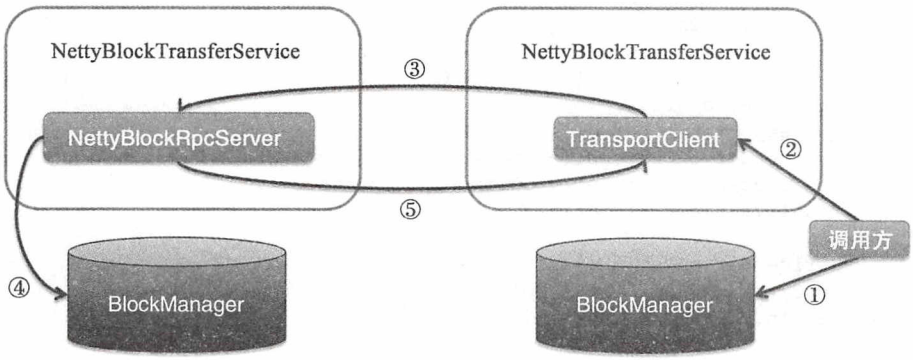


图 6-10 Block 的上传流程

序号④：NettyBlockRpcServer 接收到客户端发来的 UploadBlock 消息，将 UploadBlock 携带的 metadata 反序列化得到 lock 的存储级别及类型标记，将 UploadBlock 携带的 Block 数据（即 array）封装为 NioManagedBuffer，最后调用 BlockManager 的 putBlockData 方法将 Block 数据写入服务端本地的存储体系。

序号⑤：NettyBlockRpcServer 将处理成功的结果返回给客户端，客户端从 outstandingRpcs 中查找到本次请求对应的匿名回调类 RpcResponseCallback，并调用此匿名回调类 RpcResponseCallback 的 onSuccess 方法处理正确的响应。

4. 同步向远端上传 Block

由于 uploadBlock 是异步的，因此 NettyBlockTransferService 的父类 BlockTransferService 提供了以同步方式向远端上传 Block 的方法，其实现如代码清单 6-111 所示。

代码清单6-111 同步向远端上传Block

```
def fetchBlockSync(host: String, port: Int, execId: String, blockId: String): ManagedBuffer = {
  val result = Promise[ManagedBuffer]()
  fetchBlocks(host, port, execId, Array(blockId),
    new BlockFetchingListener {
      override def onBlockFetchFailure(blockId: String, exception: Throwable):
          Unit = {
        result.failure(exception)
      }
      override def onBlockFetchSuccess(blockId: String, data: ManagedBuffer):
          Unit = {
        val ret = ByteBuffer.allocate(data.size.toInt)
        ret.put(data.nioByteBuffer())
        ret.flip()
        result.success(new NioManagedBuffer(ret))
      }
    })
```

```
    ThreadUtils.awaitResult(result.future, Duration.Inf)
  }
```

根据代码清单 6-111，其实质为调用 uploadBlock。

6.10 DiskBlockObjectWriter 详解

我们知道，BlockManager 的 getDiskWriter 方法用于创建 DiskBlockObjectWriter。DiskBlockObjectWriter 将在 Shuffle 阶段将 map 任务的输出写入磁盘，这样 reduce 任务就能够从磁盘中获取 map 任务的中间输出了。由于 DiskBlockObjectWriter 是通过 BlockManager 获取或创建的，所以本书将 DiskBlockObjectWriter 放在本章介绍。

DiskBlockObjectWriter 用于将 JVM 中的对象直接写入磁盘文件中。DiskBlockObjectWriter 允许将数据追加到现有 Block。为了提高效率，DiskBlockObjectWriter 保留了跨多个提交的底层文件通道。

分析 DiskBlockObjectWriter，应当从 DiskBlockObjectWriter 的属性开始。DiskBlockObjectWriter 包含的构造器属性如下。

- file：要写入的文件。
- serializerManager：即 SerializerManager。
- serializerInstance：Serializer 的实例。
- bufferSize：缓冲大小。
- syncWrites：是否同步写。
- writeMetrics：类型为 ShuffleWriteMetrics，用于对 Shuffle 中间结果写入到磁盘的度量与统计。
- blockId：即块的唯一身份标识 BlockId。

DiskBlockObjectWriter 的其他属性如下。

- channel：即 FileChannel。
- mcs: 即 ManualCloseOutputStream。
- bs：即 OutputStream。
- fos：即 FileOutputStream。
- objOut：即 SerializationStream。
- ts：即 TimeTrackingOutputStream。
- initialized：是否已经初始化。
- streamOpen：是否已经打开流。
- hasBeenClosed：是否已经关闭。
- committedPosition：提交的文件位置。
- reportedPosition：报告给度量系统的文件位置。

- numRecordsWritten：已写的记录数。

了解了属性，现在就可以方便地介绍 DiskBlockObjectWriter 中提供的方法了。

1. open

DiskBlockObjectWriter 的 open 方法（见代码清单 6-112）用于打开要写入文件的各种输出流及管道。

代码清单6-112 打开DiskBlockObjectWriter

```
private def initialize(): Unit = {
  fos = new FileOutputStream(file, true)
  channel = fos.getChannel()
  ts = new TimeTrackingOutputStream(writeMetrics, fos)
  class ManualCloseBufferedOutputStream
    extends BufferedOutputStream(ts, bufferSize) with ManualCloseOutputStream
  mcs = new ManualCloseBufferedOutputStream
}

def open(): DiskBlockObjectWriter = {
  if (hasBeenClosed) {
    throw new IllegalStateException("Writer already closed. Cannot be reopened.")
  }
  if (!initialized) {
    initialize()
    initialized = true
  }

  bs = serializerManager.wrapStream(blockId, mcs)
  objOut = serializerInstance.serializeStream(bs)
  streamOpen = true
  this
}
```

2. recordWritten

DiskBlockObjectWriter 的 recordWritten 方法（见代码清单 6-113）用于对写入的记录数进行统计和度量。

代码清单6-113 写入记录数的度量

```
def recordWritten(): Unit = {
  numRecordsWritten += 1
  writeMetrics.incRecordsWritten(1)

  if (numRecordsWritten % 16384 == 0) {
    updateBytesWritten()
  }
}
private def updateBytesWritten() {
  val pos = channel.position()
```

```
    writeMetrics.incBytesWritten(pos - reportedPosition)
    reportedPosition = pos
  }
```

3. write

DiskBlockObjectWriter 的 write 方法（见代码清单 6-114）用于向输出流中写入键值对。

代码清单6-114 写入键值对

```
def write(key: Any, value: Any) {
  if (!streamOpen) {
    open()
  }

  objOut.writeKey(key)
  objOut.writeValue(value)
  recordWritten()
}
```

4. commitAndGet

DiskBlockObjectWriter 的 commitAndGet 方法（见代码清单 6-115）用于将输出流中的数据写入到磁盘。

代码清单6-115 提交并获取文件段

```
def commitAndGet(): FileSegment = {
  if (streamOpen) {
    objOut.flush()
    bs.flush()
    objOut.close()
    streamOpen = false

    if (syncWrites) {
      val start = System.nanoTime()
      fos.getFD.sync()
      writeMetrics.incWriteTime(System.nanoTime() - start)
    }

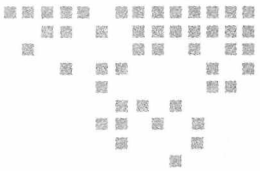
    val pos = channel.position()
    val fileSegment = new FileSegment(file, committedPosition, pos - committedPosition)
    committedPosition = pos
    writeMetrics.incBytesWritten(committedPosition - reportedPosition)
    reportedPosition = committedPosition
    fileSegment
  } else {
    new FileSegment(file, committedPosition, 0)
  }
}
```

根据代码清单 6-115，commitAndGet 方法将数据写入到文件后，创建 FileSegment 并返回。FileSegment 包含文件、写入的起始偏移量、写入的长度等信息。

6.11 小结

本章首先对构成 BlockManager 的各个组件进行了展示，然后逐一分析每个组件的实现。这些组件相互配合，进而实现了整个存储体系的功能。DiskBlockManager 和 DiskStore 一同负责磁盘存储；MemoryManager 和 MemoryStore 一同负责内存存储；BlockInfoManager 负责 Block 的元数据信息及锁的管理；BlockManager 本身则对上述组件的功能进行整合和封装；BlockManagerMaster 和 BlockManagerMasterEndpoint 一同管理分散在不同节点上的 BlockManager，BlockManagerSlaveEndpoint 则可以接收 BlockManagerMasterEndpoint 下发的各种命令；BlockTransferService 提供 Block 的上传与下载服务，Shuffle 客户端则提供 Block 的上传与下载的客户端实现。

MapOutputTracker 和 ShuffleManager 与存储体系也有一定的关系，MapOutputTracker 已在第 5 章介绍，ShuffleManager 由于与 Shuffle 的关系非常紧密，因此放在第 8 章介绍。



第7章 Chapter 7

调度系统

“诸君听吾计，使勃海引河内之众临孟津，酸枣诸将守成皋，据敖仓，塞轘辕、太谷，全制其险；使袁将军率南阳之军军丹、析，入武关，以震三辅：皆高垒深壁，勿与战，益为疑兵，示天下形势，以顺诛逆，可立定也。”

——《三国志·魏书·武帝纪》

本章导读

根据《三国志》对“联军伐卓”的记载，袁绍、袁术、韩馥、孔伷、刘岱、张邈、王匡、桥瑁、袁遗、鲍信、曹操等各方军事势力结成军事联盟，一同攻伐董卓。当时的曹操可以说是各方军事势力中最弱的，但曹操身担奋武将军一职并筹划了上面的军事调遣与分派。尽管联军并未采纳曹操的军事谋略，但是足以看出曹操是个军事天才。此后曹操从盟军中势力最弱的一方逐渐发展成为独霸北方，挟天子以令诸侯的一代枭雄。

在Spark集群上运行的任务多种多样，有的任务计算的数据量很大，有的任务执行的时间较长，有的任务执行的优先级较高，有的任务对公司的业务非常重要。整个Spark集群中各个机器的资源情况又有很多差异，有些机器CPU内核数较多，有些机器的内存很大，有些机器的磁盘空间非常充足，有些机器的网卡配置较高。有些机器在同一个机架上，有些机器可能在其他机房。任务应当分配在哪些机器上执行，任务应当分配多少资源，任务又应当有怎样的优先级？对集群资源及任务的合理分配与调度将决定整个集群的性能和效率。

本章将介绍 Spark 的调度系统，主要讲解的内容如下。

- 调度系统概述。
- 为什么需要 RDD 及 RDD 实现的初次分析。
- 阶段（Stage）。
- DAG 调度器（DAGScheduler）。
- 调度池（Pool）。
- 任务集合管理器（TaskSetManager）。
- 运行器后端接口（LauncherBackend）。
- 调度后端接口（SchedulerBackend）。
- 任务结果获取器（TaskResultGetter）。
- 任务调度器（TaskScheduler）。

7.1 调度系统概述

简单来讲，Spark 调度系统用于将用户提交的“任务”调度到集群中的不同节点执行。但是 Spark 实现的调度系统，并非一句话所概括的这么简单。Spark 资源调度分为两层：第一层是 Cluster Manager（在 YARN 模式下为 ResourceManager，在 Mesos 模式下为 Mesos Master，在 Standalone 模式下为 Master），将资源分配给 Application；第二层是 Application，进一步将资源分配给 Application 的各个 Task。第一层调度中，Standalone 模式下的 Master 对资源的调度将在 9.6.4 节详细介绍，至于其他模式中 ResourceManager、Mesos Master 对资源的调度，读者可以查阅 YARN 或 Mesos 的相关资料了解。本章所介绍的调度系统只限于第二层。

调度的对象是什么？作业或者任务。在业务场景中所说的任务与 Spark 集群中运行的任务在概念上存在很多偏差。人类往往把一件要做的事情认为是任务，因此“向 Spark 集群提交一个任务”这句话看似合情合理。工程师向 Spark 提交的一个任务，Spark 却看作一个作业（Job）。Spark 首先会对 Job 进行一系列 RDD 转换，并通过 RDD 之间的依赖关系构建有向无环图（Direct Acyclic Graph，DAG）。然后根据 RDD 依赖的不同将 RDD 划分到不同的阶段（Stage），每个阶段按照分区（Partition）的数量创建多个任务（Task）。最后将这些任务提交到集群的各个运行节点上运行。因此 Spark 中的 Task 和业务场景中所述的任务是不同的。

图 7-1 对调度系统的主要工作流程进行了展示，笔者将对此图的内容进行一些补充，以使读者更容易理解。

根据图 7-1，我们知道 Spark 调度系统主要由 DAGScheduler 和 TaskScheduler 构成。调度系统的主要工作流程如下。

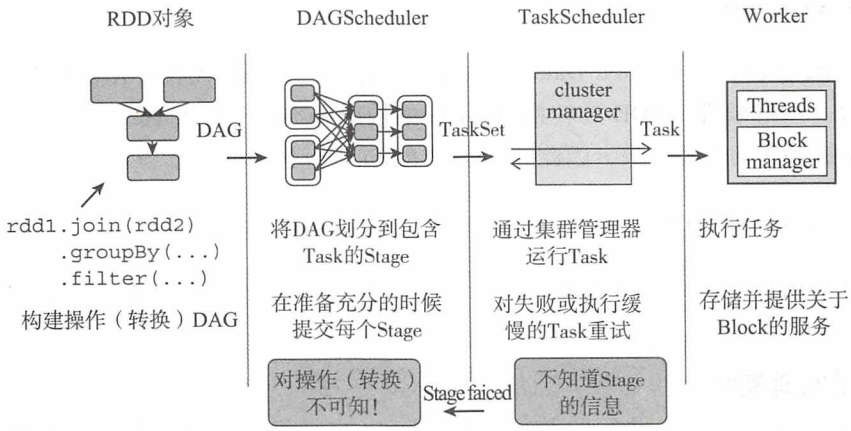


图 7-1 调度系统的主要工作流程⊖

1）build operator DAG：用户提交的 Job 将首先被转换为一系列 RDD 并通过 RDD 之间的依赖关系构建 DAG，然后将 RDD 构成的 DAG 提交到调度系统。

2）split graph into stages of tasks：DAGScheduler 负责接收由 RDD 构成的 DAG，将一系列 RDD 划分到不同的 Stage。根据 Stage 的不同类型（目前有 ResultStage 和 ShuffleMapStage 两种），给 Stage 中未完成的 Partition 创建不同类型的 Task（目前有 ResultTask 和 ShuffleMapTask 两种）。每个 Stage 将因为未完成 Partition 的多少，创建零到多个 Task。DAGScheduler 最后将每个 Stage 中的 Task 以任务集合（TaskSet）的形式提交给 Task Scheduler 继续处理。

3）launch tasks via cluster manager：使用集群管理器（cluster manager）分配资源与任务调度，对于失败的任务还会有一定的重试与容错机制。TaskScheduler 负责从 DAGScheduler 接收 TaskSet，创建 TaskSetManager 对 TaskSet 进行管理，并将此 TaskSetManager 添加到调度池中，最后将对 Task 的调度交给调度后端接口（SchedulerBackend）处理。SchedulerBackend 首先申请 TaskScheduler，按照 Task 调度算法（目前有 FIFO 和 FAIR 两种）对调度池中的所有 TaskSetManager 进行排序，然后对 TaskSet 按照最大本地性原则分配资源，最后在各个分配的节点上运行 TaskSet 中的 Task。

4）execute tasks：执行任务，并将任务中间结果和最终结果存入存储体系。


注意 以上介绍的 4 个步骤中，严格来讲只有第 2 步和第 3 步属于调度系统的范畴，第 1 步是将作业提交给调度系统前的准备工作，第 4 步也和调度系统有很多关系，例如，map 任务执行成功后将唤醒下游的 reduce 任务。

⊖ 此图源自网络，出处不可考，笔者认为画得非常好，没有必要再画了，如果原图作者看到本书，可与出版社或笔者联系。

7.2 RDD 详解

RDD（Resilient Distributed Datasets，弹性分布式数据集）代表可并行操作元素的不可变分区集合。严格来讲，RDD 的转换及 DAG 的构成并不属于调度系统的内容，但是 RDD 却是调度系统操作的主要对象，因此有必要对 RDD 进行详细介绍。

7.2.1 为什么需要 RDD

以下从数据处理模型、依赖划分原则、数据处理效率及容错处理 4 个方面解释 Spark 为什么需要 RDD。

1．数据处理模型

RDD 是一个容错的、并行的数据结构，可以控制将数据存储到磁盘或内存，能够获取数据的分区。RDD 提供了一组类似于 Scala 的操作，比如 map、flatMap、filter、reduceByKey、join、mapPartitions 等，这些操作实际是对 RDD 进行转换（transformation）。此外，RDD 还提供了 collect、foreach、count、reduce、countByKey 等操作完成数据计算的动作（action）。

当前的大数据应用场景非常丰富，如流式计算、图计算、机器学习等，它们既有相似之处，又各有不同。为了能够对所有场景下的数据处理使用统一的方式，抽象出 RDD 这一模型。

通常数据处理的模型包括迭代计算、关系查询、MapReduce、流式处理等。Hadoop 采用 MapReduce 模型，Storm 采用流式处理模型，而 Spark 则借助 RDD 实现了以上所有模型。

2. 依赖划分原则

一个 RDD 包含一个或者多个分区，每个分区实际是一个数据集合的片段。在构建 DAG 的过程中，会将 RDD 用依赖关系串联起来。每个 RDD 都有其依赖（除了最顶级 RDD 的依赖是空列表），这些依赖分为窄依赖（即 NarrowDependency）和 Shuffle 依赖（即 ShuffleDependency，也称为宽依赖）两种。为什么要对依赖进行区分？从功能角度讲它们是不一样的。NarrowDependency 会被划分到同一个 Stage 中，这样它们就能以管道的方式迭代执行。ShuffleDependency 由于所依赖的分区 Task 不止一个，所以往往需要跨节点传输数据。从容灾角度讲，它们恢复计算结果的方式不同。NarrowDependency 只需要重新执行父 RDD 的丢失分区的计算即可恢复，而 ShuffleDependency 则需要考虑恢复所有父 RDD 的丢失分区。

解释了依赖划分的原因，实际也解释了为什么要划分 Stage 这个问题。

3. 数据处理效率

RDD 的计算过程允许在多个节点并发执行。如果数据量很大，可以适当增加分区数量，这种根据硬件条件对并发任务数量的控制，能更好地利用各种资源，也能有效提高 Spark 的数据处理效率。

仅供非商业用途或交流学习使用

4. 容错处理

传统关系型数据库往往采用日志记录的方式来容灾容错，数据恢复都依赖于重新执行日志。Hadoop 为了避免单机故障概率较高的问题，通常将数据备份到其他机器容灾。由于所有备份机器同时出故障的概率比单机故障概率低很多，所以在发生宕机等问题时能够从备份机读取数据。RDD 本身是一个不可变的（Scala 中称为 immutable）数据集，当某个 Worker 节点上的 Task 失败时，可以利用 DAG 重新调度计算这些失败的 Task（执行已成功的 Task 可以从 CheckPoint（检查点）中读取，而不用重新计算）。在流式计算的场景中，Spark 需要记录日志和 CheckPoint，以便利用 CheckPoint 和日志对数据恢复。

7.2.2 RDD 实现的初次分析

有些读者可能对本节的标题感到困惑，这是因为 RDD 的 API 非常多，为了使本章对调度系统的表述更为方便，所以本节只对 RDD 中与调度系统息息相关的 API 方法进行分析，转换 API、动作 API 及检查点 API 将在 10.4 节进行介绍。

抽象类 RDD 定义了所有 RDD 的规范，我们从 RDD 的属性开始，逐步了解 RDD 的实现。

- _sc：指 SparkContext。_sc 由 @transient 修饰，所以此属性不会被序列化。
- deps：构造器参数之一，是 Dependency 的序列，用于存储当前 RDD 的依赖。RDD 的子类在实现时不一定会传递此参数。由于 deps 由 @transient 修饰，所以此属性不会被序列化。
- partitioner：当前 RDD 的分区计算器。partitioner 由 @transient 修饰，所以此属性不会被序列化。
- id：当前 RDD 的唯一身份标识。此属性通过调用 SparkContext 的 nextRddId 属性生成。
- name：RDD 的名称。name 由 @transient 修饰，所以此属性不会被序列化。
- dependencies_：与 deps 相同，但是可以被序列化。
- partitions_：存储当前 RDD 的所有分区的数组。partitions_ 由 @transient 修饰，所以此属性不会被序列化。
- storageLevel：当前 RDD 的存储级别。
- creationSite：创建当前 RDD 的用户代码。creationSite 由 @transient 修饰，所以此属性不会被序列化。
- scope：当前 RDD 的操作作用域。scope 由 @transient 修饰，所以此属性不会被序列化。
- checkpointData：当前 RDD 的检查点数据。
- checkpointAllMarkedAncestors：是否对所有标记了需要保存检查点的祖先保存检查点。
- doCheckpointCalled：是否已经调用了 doCheckpoint 方法设置检查点。此属性可以阻止对 RDD 多次设置检查点。



仅供非商业用途或交流学习使用

RDD 采用了模板方法的模式设计，抽象类 RDD 中定义了模板方法及一些未实现的接口，这些接口将需要 RDD 的各个子类分别实现。下面先来介绍 RDD 中定义的接口。

- compute：对 RDD 的分区进行计算。此方法的定义如下：

```
@DeveloperApi
def compute(split: Partition, context: TaskContext): Iterator[T]
```

- getPartitions：获取当前 RDD 的所有分区。此方法的定义如下：

```
protected def getPartitions: Array[Partition]
```

- getDependencies：获取当前 RDD 的所有依赖。此方法的定义如下：

```
protected def getDependencies: Seq[Dependency[_]] = deps
```

- getPreferredLocations：获取某一分区的偏好位置。此方法的定义如下：

```
protected def getPreferredLocations(split: Partition): Seq[String] = Nil
```

RDD 中除定义了以上接口外，还实现了一些模板方法。

1. partitions

partitions 方法（见代码清单 7-1）用于获取 RDD 的分区数组。

代码清单7-1 partitions的实现

```
final def partitions: Array[Partition] = {
  checkpointRDD.map(_.partitions).getOrElse {
    if (partitions_ == null) {
      partitions_ = getPartitions
      // 省略次要代码
    }
    partitions_
  }
}
```

根据代码清单 7-1，partitions 方法查找分区数组的优先级为：从 CheckPoint 查找→读取 partitions_ 属性→调用 getPartitions 方法获取。检查点的内容将在 10.3 节详细介绍。

2. preferredLocations

preferredLocations 方法（见代码清单 7-2）优先调用 CheckPoint 中保存的 RDD 的 getPreferredLocations 方法获取指定分区的偏好位置，当没有保存 CheckPoint 时，调用自身的 getPreferredLocations 方法获取指定分区的偏好位置。

代码清单7-2 preferredLocations的实现

```
final def preferredLocations(split: Partition): Seq[String] = {
  checkpointRDD.map(_.getPreferredLocations(split)).getOrElse {
    getPreferredLocations(split)
  }
}
```

仅供非商业用途或交流学习使用

3. dependencies

dependencies 方法（见代码清单 7-3）用于获取当前 RDD 的所有依赖的序列。

代码清单7-3 dependencies的实现

```
final def dependencies: Seq[Dependency[_]] = {
  checkpointRDD.map(r => List(new OneToOneDependency(r))).getOrElse {
    if (dependencies_ == null) {
      dependencies_ = getDependencies
    }
    dependencies_
  }
}
```

根据代码清单 7-3，dependencies 方法的执行步骤如下。

1）从 CheckPoint 中获取 RDD，并将这些 RDD 封装为 OneToOneDependency 列表。如果从 CheckPoint 中获取到 RDD 的依赖，则返回 RDD 的依赖，否则进入下一步。

2）如果 dependencies_ 等于 null，那么调用子类实现的 getDependencies 方法获取当前 RDD 的依赖后赋予 dependencies，最后返回 dependencies_。

4. 其他方法

除了以上的模板方法，RDD 中还实现了以下一些方法。

（1）context

context 方法（见代码清单 7-4）实际返回了 _sc（即 SparkContext）。



代码清单7-4 context的实现

```
def context: SparkContext = sc
```

（2）getStorageLevel

getStorageLevel 方法（见代码清单 7-5）实际返回了当前 RDD 的 StorageLevel。

代码清单7-5 获取当前RDD的存储级别

```
def getStorageLevel: StorageLevel = storageLevel
```

（3）getNarrowAncestors

getNarrowAncestors 方法（见代码清单 7-6）用于获取当前 RDD 的祖先依赖中属于窄依赖的 RDD 序列。

代码清单7-6 getNarrowAncestors的实现

```
private[spark] def getNarrowAncestors: Seq[RDD[_]] = {
  val ancestors = new mutable.HashSet[RDD[_]]
  def visit(rdd: RDD[_]) {
    val narrowDependencies = rdd.dependencies.filter(_.isInstanceOf[NarrowDependency[_]])
    val narrowParents = narrowDependencies.map(_.rdd)
    val narrowParentsNotVisited = narrowParents.filterNot(ancestors.contains)
    narrowParentsNotVisited.foreach { parent =>
```

仅供非商业用途或交流学习使用


```
        ancestors.add(parent)
        visit(parent)
      }
    }

    visit(this)

    ancestors.filterNot(_ == this).toSeq
  }
```

7.2.3 RDD 依赖

DAG 中的各个 RDD 之间存在着依赖关系。换言之，正是 RDD 之间的依赖关系构建了由 RDD 所组成的 DAG。Spark 使用 Dependency 来表示 RDD 之间的依赖关系，Dependency 的定义如下。

```
@DeveloperApi
abstract class Dependency[T] extends Serializable {
  def rdd: RDD[T]
}
```

抽象类 Dependency 只定义了一个名叫 rdd 的方法，此方法返回当前依赖的 RDD。

Dependency 分为 NarrowDependency 和 ShuffleDependency 两种依赖，下面对它们分别进行介绍。



1. 窄依赖

如果 RDD 与上游 RDD 的分区是一对一的关系，那么 RDD 和其上游 RDD 之间的依赖关系属于窄依赖（NarrowDependency）。NarrowDependency 继承了 Dependency，以表示窄依赖。NarrowDependency 的定义如下。

```
@DeveloperApi
abstract class NarrowDependency[T](_rdd: RDD[T]) extends Dependency[T] {
  def getParents(partitionId: Int): Seq[Int]
  override def rdd: RDD[T] = _rdd
}
```

NarrowDependency 定义了一个类型为 RDD 的构造器参数 _rdd，NarrowDependency 重写了 Dependency 的 rdd 方法，让其返回 _rdd。NarrowDependency 还定义了一个获取某一分区的所有父级别分区序列的 getParents 方法。NarrowDependency 一共有两个子类，它们的实现如代码清单 7-7 所示。

代码清单7-7 OneToOneDependency与RangeDependency

```
@DeveloperApi
class OneToOneDependency[T](rdd: RDD[T]) extends NarrowDependency[T](rdd) {
  override def getParents(partitionId: Int): List[Int] = List(partitionId)
}
@DeveloperApi
class RangeDependency[T](rdd: RDD[T], inStart: Int, outStart: Int, length: Int)
```

仅供非商业用途或交流学习使用


```
  extends NarrowDependency[T](rdd) {
  override def getParents(partitionId: Int): List[Int] = {
    if (partitionId >= outStart && partitionId < outStart + length) {
      List(partitionId - outStart + inStart)
    } else {
      Nil
    }
  }
}
```

根据代码清单7-7，OneToOneDependency重写的getParents方法告诉我们，子RDD的分区与依赖的父RDD的分区相同。OneToOne Dependency可以用图7-2更形象地说明。

根据代码清单7-7，RangeDependency重写了Dependency的getParents方法，其实现告诉我们Range Dependency的分区是一对一的，且索引为partitionId的子RDD分区与索引为partitionId-outStart + inStart的父RDD分区相对应（outStart代表子RDD的分区范围起始值，inStart代表父RDD的分区范围起始值）。RangeDependency可以用图7-3更形象地说明。

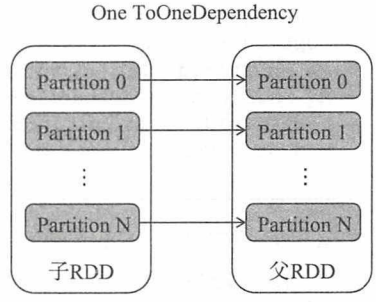


图7-2 OneToOneDependency的依赖示意图

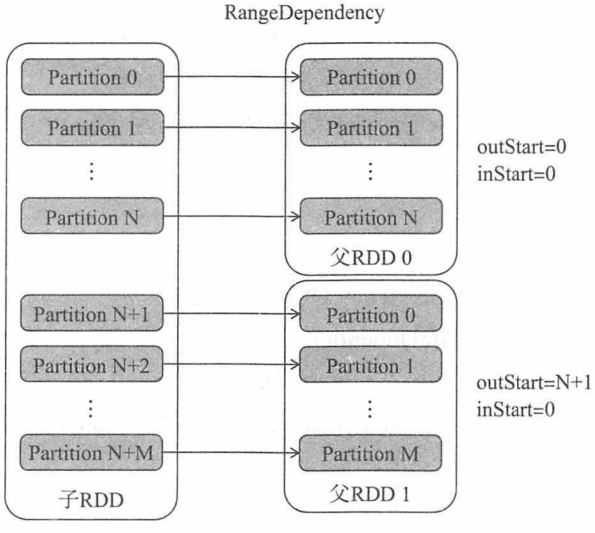


图7-3 RangeDependency的依赖示意图

2. Shuffle依赖

RDD与上游RDD的分区如果不是一对一的关系，或者RDD的分区依赖于上游RDD的多个分区，那么这种依赖关系就叫做Shuffle依赖（ShuffleDependency）。ShuffleDependency的实现如代码清单7-8所示。

仅供非商业用途或交流学习使用

代码清单7-8 ShuffleDependency的实现

```
@DeveloperApi
class ShuffleDependency[K: ClassTag, V: ClassTag, C: ClassTag](
    @transient private val _rdd: RDD[_ <: Product2[K, V]],
    val partitioner: Partitioner,
    val serializer: Serializer = SparkEnv.get.serializer,
    val keyOrdering: Option[Ordering[K]] = None,
    val aggregator: Option[Aggregator[K, V, C]] = None,
    val mapSideCombine: Boolean = false)
  extends Dependency[Product2[K, V]] {
  override def rdd: RDD[Product2[K, V]] = _rdd.asInstanceOf[RDD[Product2[K, V]]]
  private[spark] val keyClassName: String = reflect.classTag[K].runtimeClass.getName
  private[spark] val valueClassName: String = reflect.classTag[V].runtimeClass.getName
  private[spark] val combinerClassName: Option[String] =
    Option(reflect.classTag[C]).map(_.runtimeClass.getName)
  val shuffleId: Int = _rdd.context.newShuffleId()
  val shuffleHandle: ShuffleHandle = _rdd.context.env.shuffleManager.registerShuffle(
    shuffleId, _rdd.partitions.length, this)
  _rdd.sparkContext.cleaner.foreach(_.registerShuffleForCleanup(this))
}
```

根据代码清单 7-8，ShuffleDependency 定义了如下属性。

- _rdd：泛型要求必须是 Product2[K, V] 及其子类的 RDD。
- partitioner：分区计算器 Partitioner。Partitioner 将在下一节详细介绍。
- serializer：SparkEnv 中创建的 serializer，即 org.apache.spark.serializer.JavaSerializer。
- keyOrdering：按照 K 进行排序的 scala.math.Ordering 的实现类。
- aggregator：对 map 任务的输出数据进行聚合的聚合器。
- mapSideCombine：是否在 map 端进行合并，默认为 false。
- keyClassName：K 的类名。
- valueClassName：V 的类名。
- combinerClassName：结合器 C 的类名。
- shuffleId：当前 ShuffleDependency 的身份标识。
- shuffleHandle：当前 ShuffleDependency 的处理器。

此外，ShuffleDependency 还重写了父类 Dependency 的 rdd 方法，其实现将 _rdd 转换为 RDD[Product2[K, V]] 后返回。ShuffleDependency 在构造的过程中还将自己注册到 SparkContext 的 ContextCleaner 中。

7.2.4 分区计算器 Partitioner

RDD 之间的依赖关系如果是 Shuffle 依赖，那么上游 RDD 该如何确定每个分区的输出将交由下游 RDD 的哪些分区呢？或者下游 RDD 的各个分区将具体依赖于上游 RDD 的哪些分区呢？Spark 提供了分区计算器来解决这个问题。ShuffleDependency 的 partitioner 属性的类型是 Partitioner，抽象类 Partitioner 定义了分区计算器的接口规范，ShuffleDependency

仅供非商业用途或交流学习使用

的分区取决于 Partitioner 的具体实现。Partitioner 的定义如下。

```
abstract class Partitioner extends Serializable {
  def numPartitions: Int
  def getPartition(key: Any): Int
}
```

Partitioner 的 numPartitions 方法用于获取分区数量。Partitioner 的 getPartition 方法用于将输入的 key 映射到下游 RDD 的从 0 到 numPartitions-1 这一范围中的某一个分区。

Partitioner 有很多具体的实现类，它们的继承体系如图 7-4 所示。

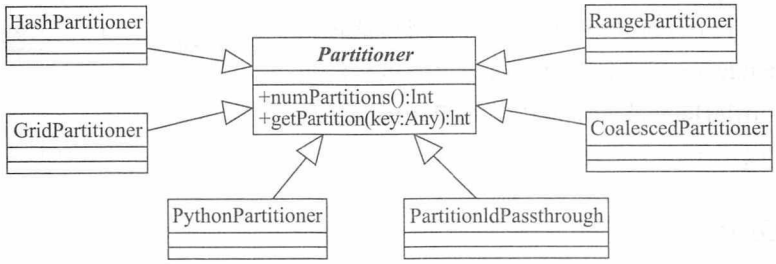


图 7-4 Partitioner 的继承体系

Spark 中除图 7-4 列出的 Partitioner 子类，还有很多 Partitioner 的匿名实现类，这里就不一一介绍了。本书以 HashPartitioner（哈希分区计算器）为例，详细介绍 Partitioner 的实现。之所以选择对 HashPartitioner 的实现进行分析，一方面是由于其实现简洁明了，读者更容易理解；另一方面通过介绍 HashPartitioner，已经足够达到本书的目的。

HashPartitioner 的实现如代码清单 7-9 所示。

代码清单7-9 哈希分区计算器的实现

```
class HashPartitioner(partitions: Int) extends Partitioner {
  require(partitions >= 0, s"Number of partitions ($partitions) cannot be
      negative.")
  def numPartitions: Int = partitions
  def getPartition(key: Any): Int = key match { // 计算出下游RDD的各个分区将具体处理哪些key
    case null => 0
    case _ => Utils.nonNegativeMod(key.hashCode, numPartitions)
  }
  override def equals(other: Any): Boolean = other match {
    case h: HashPartitioner =>
      h.numPartitions == numPartitions
    case _ =>
      false
  }
  override def hashCode: Int = numPartitions
}
```

根据代码清单 7-9，HashPartitioner 增加了一个名为 partitions 的构造器参数作为分区数，重写的 numPartitions 方法只是返回了 partitions。重写的 getPartition 方法实际以 key 的

仅供非商业用途或交流学习使用

hashCode 和 numPartitions 作为参数调用了 Utils 工具类的 nonNegativeMod 方法（具体实现可以参阅附录 A）。nonNegativeMod 方法将对 key 的 hashCode 和 numPartitions 进行取模运算，得到 key 对应的分区索引。使用哈希和取模的方式，可以方便地计算出下游 RDD 的各个分区将具体处理哪些 key。由于上游 RDD 所处理的 key 的哈希值在取模后很可能产生数据倾斜，所以 HashPartitioner 并不是一个均衡的分区计算器。

根据 HashPartitioner 的实现，我们知道 ShuffleDependency 中的分区依赖关系不再是一对一的，而是取决于 key，并且当前 RDD 的某个分区将可能依赖于 ShuffleDependency 的 RDD 的任何一个分区。经过以上分析，ShuffleDependency 采用 HashPartitioner 后的分区依赖可以用图 7-5 来表示。

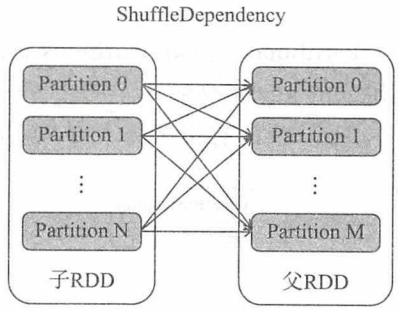


图 7-5 ShuffleDependency 的依赖示意图

7.2.5 RDDInfo

RDDInfo 用于描述 RDD 的信息，RDDInfo 提供的信息如下。

- id：RDD 的 id。
- name：RDD 的名称。
- numPartitions：RDD 的分区数量。
- storageLevel：RDD 的存储级别（即 StorageLevel）。
- parentIds：RDD 的父 RDD 的 id 序列。这说明一个 RDD 会有零到多个父 RDD。
- callSite：RDD 的用户调用栈信息。
- scope：RDD 的操作范围。scope 的类型为 RDDOperationScope，每一个 RDD 都有一个 RDDOperationScope。RDDOperationScope 与 Stage 或 Job 之间并无特殊关系，一个 RDDOperationScope 可以存在于一个 Stage 内，也可以跨越多个 Job。
- numCachedPartitions：缓存的分区数量。
- memSize：使用的内存大小。
- diskSize：使用的磁盘大小。
- externalBlockStoreSize：Block 存储在外部的大小。

RDDInfo 还提供了以下方法。

- isCached：是否已经缓存。isCached 的实现如代码清单 7-10 所示。

代码清单7-10 isCached的实现

```
def isCached: Boolean = (memSize + diskSize > 0) && numCachedPartitions > 0
```

- compare：由于 RDDInfo 继承了 Ordered，所以重写了 compare 方法用于排序。compare 的实现如代码清单 7-11 所示。

仅供非商业用途或交流学习使用

代码清单7-11 compare的实现

```
override def compare(that: RDDInfo): Int = {
  this.id - that.id
}
```

此外，RDDInfo的伴生对象中定义了fromRdd方法，用于从RDD构建出对应的RDDInfo，其实现如代码清单7-12所示。

代码清单7-12 RDDInfo伴生对象的fromRdd方法

```
private[spark] object RDDInfo {
  def fromRdd(rdd: RDD[_]): RDDInfo = {
    val rddName = Option(rdd.name).getOrElse(Utils.getFormattedClassName(rdd))
    val parentIds = rdd.dependencies.map(_.rdd.id)
    new RDDInfo(rdd.id, rddName, rdd.partitions.length,
      rdd.getStorageLevel, parentIds, rdd.creationSite.shortForm, rdd.scope)
  }
}
```

根据代码清单7-12，fromRdd方法的执行步骤如下。

1）获取当前RDD的名称（即name属性）作为RDDInfo的name属性，如果RDD还没有名称，那么调用Utils工具类的getFormattedClassName方法（见附录A）生成RDDInfo的name属性。

2）获取当前RDD依赖的所有父RDD的身份标识作为RDDInfo的parentIds属性。

3）创建RDDInfo对象。



7.3 Stage详解

DAGScheduler会将Job的RDD划分到不同的Stage，并构建这些Stage的依赖关系。这样可以使得没有依赖关系的Stage并行执行，并保证有依赖关系的Stage顺序执行。并行执行能够有效利用集群资源，提升运行效率，而串行执行则适用于那些在时间和数据资源上存在强制依赖的场景。Stage分为需要处理Shuffle的ShuffleMapStage和最下游的ResultStage。上游Stage先于下游Stage执行，ResultStage是最后执行的Stage。要了解Stage，应该从Stage的属性开始。Stage的属性如下。

- id：Stage的身份标识。
- rdd：当前Stage包含的RDD。
- numTasks：当前Stage的Task数量。
- parents：当前Stage的父Stage列表。这说明一个Stage可以有一到多个父亲Stage。
- firstJobId：第一个提交当前Stage的Job的身份标识（即Job的id）。当使用FIFO调度时，通过firstJobId首先计算来自较早Job的Stage，或者在发生故障时更快的恢复。

仅供非商业用途或交流学习使用

- callSite：应用程序中与当前 Stage 相关联的调用栈信息。
- numPartitions：当前 Stage 的分区数量。numPartitions 实际为 rdd 的分区的数量。
- jobIds：当前 Stage 所属的 Job 的身份标识集合。这说明一个 Stage 可以属于一到多个 Job。
- pendingPartitions：存储待处理分区的索引的集合。
- nextAttemptId：用于生成 Stage 下一次尝试的身份标识。
- _latestInfo：Stage 最近一次尝试的信息，即 StageInfo。
- fetchFailedAttemptIds：发生过 FetchFailure 的 Stage 尝试的身份标识的集合。此属性用于避免在发生 FetchFailure 后无止境的重试。

有了对 Stage 属性的了解，现在看看 Stage 提供的方法。

- clearFailures：清空 fetchFailedAttemptIds。
- failedOnFetchAndShouldAbort：用于将发生 FetchFailure 的 Stage 尝试的身份标识添加到 fetchFailedAttemptIds 中，并返回发生 FetchFailure 的次数是否已经超过了允许发生 FetchFailure 的次数的状态。允许发生 FetchFailure 的次数固定为 4。
- latestInfo：返回最近一次 Stage 尝试的 StageInfo，即返回 _latestInfo。
- findMissingPartitions：找到还未执行完成的分区。此方法需要子类实现。
- makeNewStageAttempt：用于创建新的 Stage 尝试，实现如代码清单 7-13 所示。



代码清单7-13　创建新的Stage尝试

```
def makeNewStageAttempt(
    numPartitionsToCompute: Int,
    taskLocalityPreferences: Seq[Seq[TaskLocation]] = Seq.empty): Unit = {
  val metrics = new TaskMetrics
  metrics.register(rdd.sparkContext)
  _latestInfo = StageInfo.fromStage(
    this, nextAttemptId, Some(numPartitionsToCompute), metrics, taskLocalityPreferences)
  nextAttemptId += 1
}
```

根据代码清单 7-13，makeNewStageAttempt 的执行步骤如下。

1）调用 StageInfo 的 fromStage 方法（见代码清单 7-19）创建新的 StageInfo。

2）增加 nextAttemptId。

抽象类 Stage 有两个实现子类，分别为 ShuffleMapStage 和 ResultStage，下面将逐个介绍。

7.3.1　ResultStage 的实现

ResultStage 可以使用指定的函数对 RDD 中的分区进行计算并得出最终结果。ResultStage 是最后执行的 Stage，此阶段主要进行作业的收尾工作（例如，对各个分区的数据收拢、打印到控制台或写入到 HDFS）。ResultStage 除继承自父类 Stage 的属性外，还包括以下属性。

- func：即对 RDD 的分区进行计算的函数。func 是 ResultStage 的构造器参数，指定了函数的形式必须满足：

```
(TaskContext, Iterator[_]) => _
```

- partitions：由 RDD 的各个分区的索引组成的数组。
- _activeJob：ResultStage 处理的 ActiveJob。

ResultStage 还提供了一些方法（见代码清单 7-14）。由于这些方法的实现都很简单，所以把它们一次性列出。

代码清单7-14 ResultStage提供的方法

```
def activeJob: Option[ActiveJob] = _activeJob
def setActiveJob(job: ActiveJob): Unit = {
  _activeJob = Option(job)
}
def removeActiveJob(): Unit = {
  _activeJob = None
}
override def findMissingPartitions(): Seq[Int] = {
  val job = activeJob.get
  (0 until job.numPartitions).filter(id => !job.finished(id))
}
```

这里特别对 findMissingPartitions 做一些解释：findMissingPartitions 用于找出当前 Job 的所有分区中还没有完成的分区的索引。ResultStage 判断一个分区是否完成，是通过 ActiveJob 的 Boolean 类型数组 finished，因为 finished 记录了每个分区是否完成。

7.3.2 ShuffleMapStage 的实现

ShuffleMapStage 是 DAG 调度流程的中间 Stage，它可以包括一到多个 ShuffleMapTask，这些 ShuffleMapTask 将生成用于 Shuffle 的数据。ShuffleMapStage 一般是 ResultStage 或者其他 ShuffleMapStage 的前置 Stage，ShuffleMapTask 则通过 Shuffle 与下游 Stage 中的 Task 串联起来。从 ShuffleMapStage 的命名可以看出，它将对 Shuffle 的数据映射到下游 Stage 的各个分区中。ShuffleMapStage 除继承自父类 Stage 的属性外，还包括以下属性。

- shuffleDep：与 ShuffleMapStage 相对应的 ShuffleDependency。
- _mapStageJobs：与 ShuffleMapStage 相关联的 ActiveJob 的列表。
- _numAvailableOutputs：ShuffleMapStage 可用的 map 任务的输出数量，这也代表了执行成功的 map 任务数。
- outputLocs：ShuffleMapStage 的各个 map 任务与其对应的 MapStatus 列表的映射关系。由于 map 任务可能会运行多次，因而可能会有多个 MapStatus。

ShuffleMapStage 还提供了一些方法，分别如下。

- mapStageJobs：即读取 _mapStageJobs 的方法。

- addActiveJob 与 removeActiveJob：向 ShuffleMapStage 相关联的 ActiveJob 的列表中添加或删除 ActiveJob。实现如代码清单 7-15 所示。

代码清单7-15 addActiveJob与removeActiveJob

```
def addActiveJob(job: ActiveJob): Unit = {
  _mapStageJobs = job :: _mapStageJobs
}

def removeActiveJob(job: ActiveJob): Unit = {
  _mapStageJobs = _mapStageJobs.filter(_ != job)
}
```

- numAvailableOutputs：即读取 _numAvailableOutputs 的方法。
- isAvailable：当 _numAvailableOutputs 与 numPartitions 相等时为 true。也就是说，ShuffleMapStage 的所有分区的 map 任务都执行成功后，ShuffleMapStage 才是可用的。
- findMissingPartitions：找到所有还未执行成功而需要计算的分区。其实现如代码清单 7-16 所示。

代码清单7-16 findMissingPartitions的实现

```
override def findMissingPartitions(): Seq[Int] = {
  val missing = (0 until numPartitions).filter(id => outputLocs(id).isEmpty)
  assert(missing.size == numPartitions - _numAvailableOutputs,
    s"${missing.size} missing, expected ${numPartitions - _numAvailableOutputs}")
  missing
}
```

- addOutputLoc：当某一分区的任务执行完成后，首先将分区与 MapStatus 的对应关系添加到 outputLocs 中，然后将可用的输出数加一。其实现如代码清单 7-17 所示。

代码清单7-17 添加分区对应的MapStatus

```
def addOutputLoc(partition: Int, status: MapStatus): Unit = {
  val prevList = outputLocs(partition)
  outputLocs(partition) = status :: prevList
  if (prevList == Nil) {
    _numAvailableOutputs += 1
  }
}
```

7.3.3 StageInfo

StageInfo 用于描述 Stage 信息，并可以传递给 SparkListener。StageInfo 包括以下属性。

- stageId：Stage 的 id。
- attemptId：当前 Stage 尝试的 id。
- name：当前 Stage 的名称。

书籍是人类进步的阶梯

- numTasks：当前 Stage 的 Task 数量。
- rddInfos：RDD 信息（即 RDDInfo）的序列。
- parentIds：当前 Stage 的父亲 Stage 的身份标识序列。
- details：详细的线程栈信息。
- taskMetrics：Task 的度量信息。
- taskLocalityPreferences：类型为 Seq[Seq[TaskLocation]]，用于存储任务的本地性偏好。
- submissionTime：DAGScheduler 将当前 Stage 提交给 TaskScheduler 的时间。
- completionTime：当前 Stage 中的所有 Task 完成的时间（即 Stage 完成的时间）或者 Stage 被取消的时间。
- failureReason：如果 Stage 失败了，用于记录失败的原因。
- accumulables：存储了所有聚合器计算的最终值。

StageInfo 提供了一个当 Stage 失败时要调用的方法 stageFailed，其实现如代码清单 7-18 所示。

代码清单7-18 stageFailed的实现

```
def stageFailed(reason: String) {
  failureReason = Some(reason)
  completionTime = Some(System.currentTimeMillis)
}
```

根据代码清单 7-18，stageFailed 方法将保存 Stage 失败的原因和 Stage 完成的时间。

在 StageInfo 的伴生对象中还提供了构建 StageInfo 的方法，如代码清单 7-19 所示。

代码清单7-19 StageInfo的fromStage方法

```
def fromStage(
    stage: Stage,
    attemptId: Int,
    numTasks: Option[Int] = None,
    taskMetrics: TaskMetrics = null,
    taskLocalityPreferences: Seq[Seq[TaskLocation]] = Seq.empty
  ): StageInfo = {
  val ancestorRddInfos = stage.rdd.getNarrowAncestors.map(RDDInfo.fromRdd)
  val rddInfos = Seq(RDDInfo.fromRdd(stage.rdd)) ++ ancestorRddInfos
  new StageInfo(
    stage.id,
    attemptId,
    stage.name,
    numTasks.getOrElse(stage.numTasks),
    rddInfos,
    stage.parents.map(_.id),
    stage.details,
    taskMetrics,
    taskLocalityPreferences)
}
```

书籍是人类进步的阶梯

根据代码清单 7-19，fromStage 方法的执行步骤如下。

1）调用当前 Stage 的 RDD 的 getNarrowAncestors 方法（见代码清单 7-6），获取 RDD 的祖先依赖中属于窄依赖的 RDD 序列。

2）对上一步中获得的 RDD 序列中的每个 RDD，调用 RDDInfo 伴生对象的 fromRdd 方法（见代码清单 7-12）创建 RDDInfo 对象。

3）给当前 Stage 的 RDD 创建对应的 RDDInfo 对象，将上一步中创建的所有 RDDInfo 对象与此 RDDInfo 对象放入序列 rddInfos 中。

4）创建 StageInfo。

7.4 面向 DAG 的调度器 DAGScheduler

DAGScheduler 实现了面向 DAG 的高层次调度，即将 DAG 中的各个 RDD 划分到不同的 Stage。DAGScheduler 可以通过计算将 DAG 中的一系列 RDD 划分到不同的 Stage，然后构建这些 Stage 之间的父子关系，最后将每个 Stage 按照 Partition 切分为多个 Task，并以 Task 集合（即 TaskSet）的形式提交给底层的 TaskScheduler。

所有的组件都通过向 DAGScheduler 投递 DAGSchedulerEvent 来使用 DAGScheduler。DAGScheduler 内部的 DAGSchedulerEventProcessLoop 将处理这些 DAGScheduler-Event，并调用 DAGScheduler 的不同方法。JobListener 用于对作业中每个 Task 执行成功或失败进行监听，JobWaiter 实现了 JobListener 并最终确定作业的成功与失败。在正式介绍 DAGScheduler 之前，我们先来看看 DAGScheduler 所依赖的组件 DAGSchedulerEventProcessLoop、JobListener 及 ActiveJob 的实现。

7.4.1 JobListener 与 JobWaiter

JobListener 定义了所有 Job 的监听器的接口规范，其定义如下。

```
private[spark] trait JobListener {
  def taskSucceeded(index: Int, result: Any): Unit
  def jobFailed(exception: Exception): Unit
}
```

Job 执行成功后将调用 JobListener 定义的 taskSucceeded 方法，而在 Job 失败后调用 JobListener 定义的 jobFailed 方法。

JobListener 有 JobWaiter 和 ApproximateActionListener 两个实现类。JobWaiter 用于等待整个 Job 执行完毕，然后调用给定的处理函数对返回结果进行处理。ApproximateActionListener 只对有单一返回结果的 Action（如 count() 和非并行的 reduce()）进行监听。本节将重点介绍 JobWaiter，ApproximateActionListener 留给感兴趣的读者自行分析。

JobWaiter 有以下成员属性。

书籍是人类进步的阶梯

- dagScheduler：即 DAGScheduler，当前 JobWaiter 等待执行完成的 Job 的调度者。
- jobId：当前 JobWaiter 等待执行完成的 Job 的身份标识。
- totalTasks：等待完成的 Job 包括的 Task 数量。
- resultHandler：执行结果的处理器。resultHandler 是 JobWaiter 构造器的一个函数参数，参数定义为：resultHandler: (Int, T) => Unit)。
- finishedTasks：等待完成的 Job 中已经完成的 Task 数量。
- jobPromise：类型为 scala.concurrent.Promise。jobPromise 用来代表 Job 完成后的结果。如果 totalTasks 等于零，说明没有 Task 需要执行，此时 jobPromise 将被直接设置为 Success。

此外，JobWaiter 定义了以下方法。

- jobFinished：Job 是否已经完成。jobFinished 的实现如下。

```
def jobFinished: Boolean = jobPromise.isCompleted
```

- completionFuture：jobPromise 的 Future。completionFuture 的实现如下。

```
def completionFuture: Future[Unit] = jobPromise.future
```

- cancel：取消对 Job 的执行。cancel 的实现如下。

```
def cancel() {
  dagScheduler.cancelJob(jobId)
}
```

根据上述代码，cancel 实际调用了 DAGScheduler 的 cancelJob 方法。DAGScheduler 的 cancelJob 方法将使得 Job 的所有任务被取消，并向外抛出 SparkException。

- taskSucceeded：JobWaiter 重写的在特质 JobListener 中定义的方法，实现如代码清单 7-20 所示。

代码清单7-20 taskSucceeded的实现

```
override def taskSucceeded(index: Int, result: Any): Unit = {
  synchronized {
    resultHandler(index, result.asInstanceOf[T])
  }
  if (finishedTasks.incrementAndGet() == totalTasks) {
    jobPromise.success(())
  }
}
```

根据代码清单 7-20，taskSucceeded 的执行步骤如下。

1）调用 resultHandler 函数来处理 Job 中每个 Task 的执行结果。

2）增加已完成的 Task 数量。

3）如果 Job 的所有 Task 都已经完成，那么将 jobPromise 设置为 Success。

- jobFailed：JobWaiter 重写的由 JobListener 定义的方法，实现如代码清单 7-21 所示。

书籍是人类进步的阶梯

代码清单7-21 jobFailed的实现

```
override def jobFailed(exception: Exception): Unit = {
  if (!jobPromise.tryFailure(exception)) {
    logWarning("Ignore failure", exception)
  }
}
```

根据代码清单 7-21，jobFailed 实际只是将 jobPromise 设置为 Failure。

7.4.2 ActiveJob 详解

ActiveJob 用来表示已经激活的 Job，即被 DAGScheduler 接收处理的 Job。ActiveJob 有以下属性。

- jobId：Job 的身份标识。
- finalStage：Job 的最下游 Stage。
- callSite：应用程序调用栈。
- listener：监听当前 Job 的 JobListener。
- properties：包含了当前 Job 的调度、Job group、描述等属性的 Properties。
- numPartitions：当前 Job 的分区数量。如果 finalStage 为 ResultStage，那么此属性等于 ResultStage 的 partitions 属性的长度。如果 finalStage 为 ShuffleMapStage，那么此属性等于 ShuffleMapStage 的 rdd 的 partitions 属性的长度。
- finished：Boolean 类型的数组，每个数组索引代表一个分区的任务是否执行完成。
- numFinished：当前 Job 的所有任务中已完成任务的数量。

7.4.3 DAGSchedulerEventProcessLoop 的简要介绍

DAGSchedulerEventProcessLoop 是 DAGScheduler 内部的事件循环处理器，用于处理 DAGSchedulerEvent 类型的事件。由于 DAGSchedulerEventProcessLoop 的实现与 LiveListenerBus 非常相似，而且其实现更为简单，所以我们将省略对 DAGSchedulerEventProcessLoop 的原理分析，只关注 DAGSchedulerEventProcessLoop 能够处理哪些事件。DAGS chedulerEventProcessLoop 能够处理的事件类型如代码清单 7-22 所示。

代码清单7-22 DAGSchedulerEventProcessLoop处理的消息类型

```
private def doOnReceive(event: DAGSchedulerEvent): Unit = event match {
  case JobSubmitted(jobId, rdd, func, partitions, callSite, listener, properties) =>
    dagScheduler.handleJobSubmitted(jobId, rdd, func, partitions, callSite,
        listener, properties)
  case MapStageSubmitted(jobId, dependency, callSite, listener, properties) =>
    dagScheduler.handleMapStageSubmitted(jobId, dependency, callSite, listener,
        properties)
  case StageCancelled(stageId) =>
    dagScheduler.handleStageCancellation(stageId)
  case JobCancelled(jobId) =>
```

书籍是人类进步的阶梯

```
      dagScheduler.handleJobCancellation(jobId)
    case JobGroupCancelled(groupId) =>
      dagScheduler.handleJobGroupCancelled(groupId)
    case AllJobsCancelled =>
      dagScheduler.doCancelAllJobs()
    case ExecutorAdded(execId, host) =>
      dagScheduler.handleExecutorAdded(execId, host)
    case ExecutorLost(execId, reason) =>
      val filesLost = reason match {
        case SlaveLost(_, true) => true
        case _ => false
      }
      dagScheduler.handleExecutorLost(execId, filesLost)
    case BeginEvent(task, taskInfo) =>
      dagScheduler.handleBeginEvent(task, taskInfo)
    case GettingResultEvent(taskInfo) =>
      dagScheduler.handleGetTaskResult(taskInfo)
    case completion: CompletionEvent =>
      dagScheduler.handleTaskCompletion(completion)
    case TaskSetFailed(taskSet, reason, exception) =>
      dagScheduler.handleTaskSetFailed(taskSet, reason, exception)
    case ResubmitFailedStages =>
      dagScheduler.resubmitFailedStages()
  }
```

7.4.4 DAGScheduler 的组成

DAGScheduler 本身包含很多成员，了解这些成员是深入理解 DAGScheduler 的前提。DAGScheduler 的成员包括以下这些。

- sc：SparkContext。
- taskScheduler：TaskScheduler 的引用。
- listenerBus：LiveListenerBus。
- mapOutputTracker：MapOutputTrackerMaster。
- blockManagerMaster：BlockManagerMaster。
- env：SparkEnv。
- clock：时钟对象。类型为 SystemClock，提供了获取系统当前毫秒时间的 getTimeMillis 方法和使得当前线程睡眠直到指定时间为止的 waitTillTime 方法。SystemClock 的实现很简单，所以这里不详细说明，感兴趣的读者可自行了解。
- metricsSource：有关 DAGScheduler 的度量源（即 DAGSchedulerSource）。
- nextJobId：类型为 AtomicInteger，用于生成下一个 Job 的身份标识（即 JobId）。
- numTotalJobs：总共提交的作业数量。numTotalJobs 实际读取了 nextJobId 的当前值。
- nextStageId：类型为 AtomicInteger，用于生成下一个 Stage 的身份标识（即 StageId）。
- jobIdToStageIds：用于缓存 JobId 与 StageId 之间的映射关系。由于 jobIdToStageIds 的类型为 HashMap[Int, HashSet[Int]]，所以 Job 与 Stage 之间是一对多的关系。
- stageIdToStage：用于缓存 StageId 与 Stage 之间的映射关系。

书籍是人类进步的阶梯

- shuffleIdToMapStage：用于缓存 Shuffle 的身份标识（即 shuffleId）与 ShuffleMapStage 之间的映射关系。
- jobIdToActiveJob：Job 的身份标识与激活的 Job（即 ActiveJob）之间的映射关系。
- waitingStages：处于等待状态的 Stage 集合。
- runningStages：处于运行状态的 Stage 集合。
- failedStages：处于失败状态的 Stage 集合。
- activeJobs：所有激活的 Job 的集合。
- cacheLocs：缓存每个 RDD 的所有分区的位置信息。cacheLocs 的数据类型是 HashMap[Int, IndexedSeq[Seq[TaskLocation]]]，所以每个 RDD 的分区按照分区号作为索引存储到 IndexedSeq。由于 RDD 的每个分区作为一个 Block 以及存储体系的复制因素，因此 RDD 的每个分区的 Block 可能存在于多个节点的 BlockManager 上，RDD 每个分区的位置信息为 TaskLocation 的序列。
- failedEpoch：当检测到一个节点出现故障时，会将执行失败的 Executor 和 MapOutputTracker 当前的纪元添加到 failedEpoch。此外，还使用 failedEpoch 忽略迷失的 ShuffleMapTask 的结果。
- outputCommitCoordinator：SparkEnv 的子组件 OutputCommitCoordinator。
- closureSerializer：SparkEnv 中创建的 closureSerializer。
- disallowStageRetryForTest：在测试过程中，当发生 FetchFailed 时，使用此属性则不会对 Stage 进行重试。可以通过 spark.test.noStageRetry 进行属性配置，默认为 false。
- messageScheduler：只有一个线程的 ScheduledThreadPoolExecutor，创建的线程以 dag-scheduler-message 开头。messageScheduler 的职责是对失败的 Stage 进行重试。
- eventProcessLoop：DAGSchedulerEventProcessLoop。

7.4.5 DAGScheduler 提供的常用方法

上一节我们对 DAGScheduler 中的各个成员有了基本的了解，这一节来介绍 DAGScheduler 提供的常用方法。

1. clearCacheLocs

clearCacheLocs（见代码清单 7-23）用于清空 cacheLocs 中缓存的各个 RDD 的所有分区的位置信息。

代码清单7-23 清空cacheLocs

```
private def clearCacheLocs(): Unit = cacheLocs.synchronized {
  cacheLocs.clear()
}
```

2. updateJobIdStageIdMaps

updateJobIdStageIdMaps 方法（见代码清单 7-24）用来更新 Job 的身份标识与 Stage 及

书籍是人类进步的阶梯

其所有祖先的映射关系。

代码清单7-24 updateJobIdStageIdMaps的实现

```
private def updateJobIdStageIdMaps(jobId: Int, stage: Stage): Unit = {
  @tailrec
  def updateJobIdStageIdMapsList(stages: List[Stage]) {
    if (stages.nonEmpty) {
      val s = stages.head
      s.jobIds += jobId
      jobIdToStageIds.getOrElseUpdate(jobId, new HashSet[Int]()) += s.id
      val parentsWithoutThisJobId = s.parents.filter { ! _.jobIds.contains(jobId) }
      updateJobIdStageIdMapsList(parentsWithoutThisJobId ++ stages.tail)
    }
  }
  updateJobIdStageIdMapsList(List(stage))
}
```

根据代码清单7-24，updateJobIdStageIdMaps方法对Stage及Stage的所有祖先Stage进行如下处理。

1）将jobId添加到每个Stage的jobIds。

2）将jobId和与每个Stage的id之间的映射关系更新到jobIdToStageIds。

3. activeJobForStage

activeJobForStage方法（见代码清单7-25）用于找到Stage的所有已经激活的Job的身份标识。

代码清单7-25 activeJobForStage的实现

```
private def activeJobForStage(stage: Stage): Option[Int] = {
  val jobsThatUseStage: Array[Int] = stage.jobIds.toArray.sorted
  jobsThatUseStage.find(jobIdToActiveJob.contains)
}
```

4. getCacheLocs

getCacheLocs方法（见代码清单7-26）用于获取RDD各个分区的TaskLocation序列。

代码清单7-26 获取RDD各个分区的TaskLocation序列

```
private[scheduler]
def getCacheLocs(rdd: RDD[_]): IndexedSeq[Seq[TaskLocation]] = cacheLocs.
synchronized {
  if (!cacheLocs.contains(rdd.id)) {
    val locs: IndexedSeq[Seq[TaskLocation]] = if (rdd.getStorageLevel ==
       StorageLevel.NONE) {
      IndexedSeq.fill(rdd.partitions.length)(Nil)
    } else {
      val blockIds =
        rdd.partitions.indices.map(index => RDDBlockId(rdd.id, index)).
            toArray[BlockId]
      blockManagerMaster.getLocations(blockIds).map { bms => // 获取每个RDDBlockId
          所存储的位置信息
```

```
            bms.map(bm => TaskLocation(bm.host, bm.executorId))
          }
        }
        cacheLocs(rdd.id) = locs
      }
      cacheLocs(rdd.id)
    }
```

根据代码清单 7-26，getCacheLocs 方法的执行步骤如下。

1）如果 cacheLocs 中不包含 RDD 对应的 IndexedSeq[Seq[TaskLocation]]，那么执行以下操作：

① 如果 RDD 的存储级别为 NONE，那么构造一个空的 IndexedSeq[Seq[TaskLocation]] 返回。

② 如果 RDD 的存储级别不为 NONE，则首先构造 RDD 各个分区的 RDDBlockId 数组，然后调用 BlockManagerMaster 的 getLocations 方法获取数组中每个 RDDBlockId 所存储的位置信息（即 BlockManager 的身份标识 BlockManagerId）序列，并封装为 TaskLocation。

③ 将 RDD 的 id 与第①或第②步得到的 IndexedSeq[Seq[TaskLocation]] 之间的映射关系放入 cacheLocs。

2）返回 cacheLocs 中缓存的 RDD 对应的 IndexedSeq[Seq[TaskLocation]]。

5. getPreferredLocsInternal

getPreferredLocsInternal 方法（见代码清单 7-27）用于获取 RDD 的指定分区的偏好位置。

代码清单7-27 getPreferredLocsInternal的实现

```
private def getPreferredLocsInternal(
    rdd: RDD[_],
    partition: Int,
    visited: HashSet[(RDD[_], Int)]): Seq[TaskLocation] = {
  if (!visited.add((rdd, partition))) { // 避免对RDD的指定分区的重复访问
    return Nil
  }
  val cached = getCacheLocs(rdd)(partition) // 获取RDD的指定分区的位置信息
  if (cached.nonEmpty) {
    return cached
  }
  val rddPrefs = rdd.preferredLocations(rdd.partitions(partition)).toList//获取RDD
      指定分区的偏好位置
  if (rddPrefs.nonEmpty) {
    return rddPrefs.map(TaskLocation(_))
  }
  rdd.dependencies.foreach { // 以窄依赖的RDD的同一分区的偏好位置作为当前RDD的此分区的偏
      好位置
    case n: NarrowDependency[_] =>
      for (inPart <- n.getParents(partition)) {
        val locs = getPreferredLocsInternal(n.rdd, inPart, visited)
        if (locs != Nil) {
```

```
            return locs
          }
        }
      case _ =>
    }
    Nil
  }
```

根据代码清单 7-27，getPreferredLocsInternal 的执行步骤如下。

1）避免对 RDD 的指定分区的重复访问。

2）调用 getCacheLocs 方法，从 cacheLocs 中获取 RDD 的指定分区的位置信息。如果能够获取到位置信息，则返回获取到的 TaskLocation 序列，否则进入下一步。

3）调用 RDD 的 preferredLocations 方法获取指定分区的偏好位置信息，并封装为 TaskLocation 后返回。如果未能获取到位置信息，则进入下一步。

4）遍历 RDD 的窄依赖，获取窄依赖中 RDD 的同一分区的偏好位置，以首先找到的偏好位置作为当前 RDD 的偏好位置返回。

5）如果以上步骤都未能获取到 RDD 的指定分区的偏好位置，则返回 Nil。

6. getPreferredLocs

DAGScheduler 的 getPreferredLocs 方法（见代码清单 7-28）用于获取 RDD 的指定分区的偏好位置。

代码清单7-28 获取RDD的指定分区的偏好位置

```
private[spark]
def getPreferredLocs(rdd: RDD[_], partition: Int): Seq[TaskLocation] = {
  getPreferredLocsInternal(rdd, partition, new HashSet)
}
```

根据代码清单 7-28，getPreferredLocs 实际调用了 getPreferredLocsInternal 方法。

7. handleExecutorAdded

DAGScheduler 的 handleExecutorAdded 方法（见代码清单 7-29）用于将 Executor 的身份标识从 failedEpoch 中移除。

代码清单7-29 handleExecutorAdded的实现

```
private[scheduler] def handleExecutorAdded(execId: String, host: String) {
  if (failedEpoch.contains(execId)) {
    logInfo("Host added was in lost list earlier: " + host)
    failedEpoch -= execId
  }
}
```

8. executorAdded

DAGScheduler 的 executorAdded 方法（见代码清单 7-30）用于向 DAGScheduler 的 DAGScheduler-EventProcessLoop 投递 ExecutorAdded 事件。

代码清单7-30 executorAdded的实现

```
def executorAdded(execId: String, host: String): Unit = {
  eventProcessLoop.post(ExecutorAdded(execId, host))
}
```

根据代码清单7-22，我们知道DAGSchedulerEventProcessLoop在接收到ExecutorAdded事件后将调用DAGScheduler的handleExecutorAdded方法。

有了对DAGScheduler中各个成员和常用方法的了解，现在来介绍DAGScheduler的调度流程。通过对DAGScheduler的调度流程的分析，我们能进一步认识DAGScheduler提供的其他方法的实现。

7.4.6 DAGScheduler与Job的提交

1. 提交Job

用户提交的Job首先会被转换为一系列RDD，然后才交给DAGScheduler进行处理。DAGScheduler的runJob方法是这一过程的入口，其实现如代码清单7-31所示。

代码清单7-31 DAGScheduler的调度入口

```
def runJob[T, U](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    callSite: CallSite,
    resultHandler: (Int, U) => Unit,
    properties: Properties): Unit = {
  val start = System.nanoTime
  val waiter = submitJob(rdd, func, partitions, callSite, resultHandler,
      properties) // 提交Job
  val awaitPermission = null.asInstanceOf[scala.concurrent.CanAwait]
  waiter.completionFuture.ready(Duration.Inf)(awaitPermission) // 利用JobWaiter等
      待Job处理完毕
  waiter.completionFuture.value.get match { // JobWaiter监听到Job的处理结果，进行进一
      步处理
    case scala.util.Success(_) =>
      logInfo("Job %d finished: %s, took %f s".format
        (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
    case scala.util.Failure(exception) =>
      logInfo("Job %d failed: %s, took %f s".format
        (waiter.jobId, callSite.shortForm, (System.nanoTime - start) / 1e9))
      val callerStackTrace = Thread.currentThread().getStackTrace.tail
      exception.setStackTrace(exception.getStackTrace ++ callerStackTrace)
      throw exception
  }
}
```

根据代码清单7-31，runJob的执行步骤如下。

1）生成运行Job的启动时间start。

2）调用submitJob方法提交Job。由于执行Job的过程是异步的，因此submitJob方法

将立即返回 JobWaiter 对象。

3）利用 JobWaiter 等待 Job 处理完毕。如果 Job 执行成功，根据处理结果打印相应的日志。如果 Job 执行失败，除打印日志外，还将抛出引起 Job 失败的异常信息。

DAGScheduler 的 submitJob 方法用于提交 Job，其实现如代码清单 7-32 所示。

代码清单7-32 提交Job

```
def submitJob[T, U](
    rdd: RDD[T],
    func: (TaskContext, Iterator[T]) => U,
    partitions: Seq[Int],
    callSite: CallSite,
    resultHandler: (Int, U) => Unit,
    properties: Properties): JobWaiter[U] = {
  val maxPartitions = rdd.partitions.length // 获取当前Job的最大分区数maxPartitions
  partitions.find(p => p >= maxPartitions || p < 0).foreach { p =>
    throw new IllegalArgumentException(
      "Attempting to access a non-existent partition: " + p + ". " +
        "Total number of partitions: " + maxPartitions)
  }
  val jobId = nextJobId.getAndIncrement() // 生成下一个Job的jobId
  if (partitions.size == 0) {
    return new JobWaiter[U](this, jobId, 0, resultHandler)
  }
  assert(partitions.size > 0)
  val func2 = func.asInstanceOf[(TaskContext, Iterator[_]) => _]
  // 创建等待Job完成的JobWaiter
  val waiter = new JobWaiter(this, jobId, partitions.size, resultHandler)
  // 向DAGSchedulerEventProcessLoop发送JobSubmitted事件
  eventProcessLoop.post(JobSubmitted(
    jobId, rdd, func2, partitions.toArray, callSite, waiter,
    SerializationUtils.clone(properties)))
  waiter
}
```

根据代码清单 7-32，submitJob 的处理步骤如下。

1）调用 RDD 的 partitions 方法来获取当前 Job 的最大分区数 maxPartitions。如果检查到有不存在的分区，那么抛出 IllegalArgumentException。

2）生成下一个 Job 的 jobId。

3）如果 Job 的分区数量等于 0，则创建一个 totalTasks 属性为 0 的 JobWaiter 并返回。根据 JobWaiter 的实现，JobWaiter 的 jobPromise 将被设置为 Success。

4）如果 Job 的分区数量大于 0，则创建真正等待 Job 执行完成的 JobWaiter。

5）向 eventProcessActor（即 DAGSchedulerEventProcessLoop）发送 JobSubmitted 事件。

6）返回 JobWaiter。

2. 处理 Job 的提交

根据 DAGSchedulerEventProcessLoop 的实现（见代码清单 7-22），我们知道 DAGSchedulerEventProcessLoop 接收到 JobSubmitted 事件后，将调用 DAGScheduler 的 handleJob-

Submitted 方法。handleJobSubmitted 的实现如代码清单 7-33 所示。

代码清单7-33 handleJobSubmitted的实现

```
private[scheduler] def handleJobSubmitted(jobId: Int,
    finalRDD: RDD[_],
    func: (TaskContext, Iterator[_]) => _,
    partitions: Array[Int],
    callSite: CallSite,
    listener: JobListener,
    properties: Properties) {
  var finalStage: ResultStage = null
  try {
    //创建ResultStage
    finalStage = createResultStage(finalRDD, func, partitions, jobId, callSite)
  } catch {
    case e: Exception =>
      logWarning("Creating new stage failed due to exception - job: " + jobId, e)
      listener.jobFailed(e)
      return
  }

  // 创建ActiveJob
  val job = new ActiveJob(jobId, finalStage, callSite, listener, properties)
  clearCacheLocs()

  val jobSubmissionTime = clock.getTimeMillis() // 生成Job提交的时间
  jobIdToActiveJob(jobId) = job
  activeJobs += job
  finalStage.setActiveJob(job)
  val stageIds = jobIdToStageIds(jobId).toArray
  val stageInfos = stageIds.flatMap(id => stageIdToStage.get(id).map(_.latestInfo))
  listenerBus.post(
    SparkListenerJobStart(job.jobId, jobSubmissionTime, stageInfos, properties))
  submitStage(finalStage) // 提交ResultStage
}
```

根据代码清单 7-33，handleJobSubmitted 的执行过程如下。

1）调用 createResultStage 方法（见代码清单 7-34）创建 ResultStage。创建 Stage 的过程可能发生异常（比如运行在 HadoopRDD 上的任务所依赖的底层 HDFS 文件被删除了），当异常发生时需要主动调用 JobWaiter 的 jobFailed 方法。createResultStage 方法在创建 ResultStage 的过程中会引起创建一系列 Stage 的连锁反应，Job 与这些 Stage 的关系将被放入 jobIdToStageIds 中。

2）创建 ActiveJob。

3）调用 clearCacheLocs 方法（见代码清单 7-23）清空 cacheLocs。

4）生成 Job 提交的时间。

5）将 jobId 与刚创建的 ActiveJob 之间的对应关系放入 jobIdToActiveJob 中。

6）将刚创建的 ActiveJob 放入 activeJobs 集合中。

7）使 ResultStage 的 _activeJob 属性持有刚创建的 ActiveJob。

8）获取当前 Job 的所有 Stage 对应的 StageInfo（即数组 stageInfos）。

9）向 LiveListenerBus 投递 SparkListenerJobStart 事件，进而引发所有关注此事件的监听器执行相应的操作。

10）调用 submitStage 方法（见代码清单 7-40）提交 ResultStage。

7.4.7 构建 Stage

在 Spark 中，一个 Job 可能被划分为一到多个 Stage。各个 Stage 之间存在着依赖关系，下游的 Stage 依赖于上游的 Stage。Job 中所有 Stage 的提交过程包括反向驱动与正向提交。

所谓 Stage 的反向驱动，就是从最下游的 ResultStage 开始，由 ResultStage 驱动所有父 Stage 的执行，父 Stage 又驱动祖父 Stage 的执行，这个驱动过程不断向祖先方向传递，直到最上游的 Stage 为止，整个驱动过程可以说是“长江后浪推前浪”。而正向提交，就是前代 Stage 先于后代 Stage 将 Task 提交给 TaskScheduler，祖父 Stage 先于父 Stage 将 Task 提交给 TaskScheduler，父 Stage 先于子 Stage 将 Task 提交给 TaskScheduler，ResultStage 最后一个将 Task 提交给 TaskScheduler，整个提交过程可以说是“代代相传”。

本节将详细介绍 Stage 依赖关系的构建过程。

1. 创建 ResultStage

DAGScheduler 的 createResultStage 方法用于创建 ResultStage，其实现如代码清单 7-34 所示。

代码清单7-34 创建ResultStage

```
private def createResultStage(
    rdd: RDD[_],
    func: (TaskContext, Iterator[_]) => _,
    partitions: Array[Int],
    jobId: Int,
    callSite: CallSite): ResultStage = {
  val parents = getOrCreateParentStages(rdd, jobId) // 获取所有父Stage的列表
  val id = nextStageId.getAndIncrement() // 生成ResultStage的身份标识
  // 创建ResultStage
  val stage = new ResultStage(id, rdd, func, partitions, parents, jobId, callSite)
  stageIdToStage(id) = stage
  updateJobIdStageIdMaps(jobId, stage)
  stage
}
```

根据代码清单 7-34，createResultStage 的处理步骤如下。

1）调用 getOrCreateParentStages 方法（见代码清单 7-35）获取所有父 Stage 的列表，父 Stage 主要是宽依赖（ShuffleDependency）对应的 Stage，此列表内的 Stage 包含以下几种：

① 当前 RDD 的直接或间接的依赖是 ShuffleDependency 且已经注册过的 Stage。

② 当前 RDD 的直接或间接的依赖是 ShuffleDependency 且没有注册过 Stage 的。对于这种 ShuffleDependency，则根据 ShuffleDependency 中的 RDD，找到它的直接或间接的依

赖是 ShuffleDependency 且没有注册过 Stage 的所有 ShuffleDependency，为它们创建并注册 Stage。

③当前 RDD 的直接或间接的依赖是 ShuffleDependency 且没有注册过 Stage 的。为此 ShuffleDependency 创建并注册 Stage。

2）生成 Stage 的身份标识，并创建 ResultStage。

3）将 ResultStage 注册到 stageIdToStage 中。

4）调用 updateJobIdStageIdMaps 方法（见代码清单 7-24）更新 Job 的身份标识与 ResultStage 及其所有祖先的映射关系。

2. 获取或创建父 Stage 列表

Spark 中的 Job 可能包含一到多个 Stage，这些 Stage 的划分是从 ResultStage 开始，从后往前边划分边创建的。DAGScheduler 的 getOrCreateParentStages 方法（见代码清单 7-35）用于获取或者创建给定 RDD 的所有父 Stage，这些 Stage 将被分配给 jobId 对应的 Job。

代码清单7-35　获取或创建给定RDD的所有父Stage

```
private def getOrCreateParentStages(rdd: RDD[_], firstJobId: Int): List[Stage] = {
  getShuffleDependencies(rdd).map { shuffleDep =>
    getOrCreateShuffleMapStage(shuffleDep, firstJobId)
  }.toList
}
```

根据代码清单 7-35，getOrCreateParentStages 的处理步骤如下。

1）调用 DAGScheduler 的 getShuffleDependencies 方法（见代码清单 7-36）获取 RDD 的所有 ShuffleDependency 的序列，逐个访问每个 RDD 及其依赖的非 Shuffle 的 RDD，获取所有非 Shuffle 的 RDD 的 ShuffleDependency 依赖。

2）调用 DAGScheduler 的 getOrCreateShuffleMapStage 方法（见代码清单 7-37）为每一个 ShuffleDependency 获取或者创建对应的 ShuffleMapStage，并返回得到的 Shuffle-MapStage 列表。

代码清单7-36　获取RDD的所有ShuffleDependency的序列

```
private[scheduler] def getShuffleDependencies(
    rdd: RDD[_]): HashSet[ShuffleDependency[_, _, _]] = {
  val parents = new HashSet[ShuffleDependency[_, _, _]]
  val visited = new HashSet[RDD[_]]
  val waitingForVisit = new Stack[RDD[_]]
  waitingForVisit.push(rdd)
  while (waitingForVisit.nonEmpty) {
    val toVisit = waitingForVisit.pop()
    if (!visited(toVisit)) {
      visited += toVisit
      toVisit.dependencies.foreach {
        case shuffleDep: ShuffleDependency[_, _, _] =>
          parents += shuffleDep
        case dependency =>
```

```
            waitingForVisit.push(dependency.rdd)
          }
        }
      }
      parents
    }
```

getOrCreateShuffleMapStage 方法（见代码清单 7-37）用于获取或者创建 ShuffleMapStage。

代码清单7-37 获取或创建ShuffleMapStage

```
  private def getOrCreateShuffleMapStage(
      shuffleDep: ShuffleDependency[_, _, _],
      firstJobId: Int): ShuffleMapStage = {
    shuffleIdToMapStage.get(shuffleDep.shuffleId) match {
      case Some(stage) =>
          stage // 如果已经创建了ShuffleDependency对应的ShuffleMapStage，则直接返回此
  ShuffleMapStage

      case None =>
        getMissingAncestorShuffleDependencies(shuffleDep.rdd).foreach { dep =>
          if (!shuffleIdToMapStage.contains(dep.shuffleId)) {
            createShuffleMapStage(dep, firstJobId) // 创建当前Stage的上游ShuffleMapStage
          }
        }
        createShuffleMapStage(shuffleDep, firstJobId) // 为当前ShuffleDependency创建
            Shuffle MapStage
    }
  }
```

根据代码清单 7-37，getOrCreateShuffleMapStage 的处理步骤如下。

1）如果已经创建了 ShuffleDependency 对应的 ShuffleMapStage，则直接返回此 ShuffleMapStage。

2）否则调用 getMissingAncestorShuffleDependencies 方法（见代码清单 7-38）找到所有还未创建过 ShuffleMapStage 的祖先 ShuffleDependency，并调用 createShuffleMapStage 方法（见代码清单 7-39）创建 ShuffleMapStage 并注册。最后还会为当前 ShuffleDependency 调用方法 createShuffleMapStage 创建 ShuffleMapStage 并注册。

代码清单7-38 找到所有还未创建过ShuffleMapStage的祖先ShuffleDependency

```
  private def getMissingAncestorShuffleDependencies(
      rdd: RDD[_]): Stack[ShuffleDependency[_, _, _]] = {
    val ancestors = new Stack[ShuffleDependency[_, _, _]]
    val visited = new HashSet[RDD[_]]
    val waitingForVisit = new Stack[RDD[_]]
    waitingForVisit.push(rdd)
    while (waitingForVisit.nonEmpty) {
      val toVisit = waitingForVisit.pop()
      if (!visited(toVisit)) {
        visited += toVisit
        getShuffleDependencies(toVisit).foreach { shuffleDep =>
          if (!shuffleIdToMapStage.contains(shuffleDep.shuffleId)) {
```

```
            ancestors.push(shuffleDep)
            waitingForVisit.push(shuffleDep.rdd)
          } // Otherwise, the dependency and its ancestors have already been registered.
        }
      }
    }
    ancestors
  }
```

createShuffleMapStage 用于创建 ShuffleMapStage，其实现如代码清单 7-39 所示。

代码清单7-39 创建ShuffleMapStage

```
def createShuffleMapStage(shuffleDep: ShuffleDependency[_, _, _], jobId: Int):
    ShuffleMapStage = {
  val rdd = shuffleDep.rdd
  val numTasks = rdd.partitions.length
  val parents = getOrCreateParentStages(rdd, jobId)
  val id = nextStageId.getAndIncrement()
  val stage = new ShuffleMapStage(id, rdd, numTasks, parents, jobId, rdd.
      creationSite, shuffleDep)

  stageIdToStage(id) = stage
  shuffleIdToMapStage(shuffleDep.shuffleId) = stage
  updateJobIdStageIdMaps(jobId, stage)

  if (mapOutputTracker.containsShuffle(shuffleDep.shuffleId)) {
    val serLocs = mapOutputTracker.getSerializedMapOutputStatuses(shuffleDep.shuffleId)
    val locs = MapOutputTracker.deserializeMapStatuses(serLocs)
    (0 until locs.length).foreach { i =>
      if (locs(i) ne null) {
        stage.addOutputLoc(i, locs(i)) // 添加每个分区的输出位置信息
      }
    }
  } else {
    logInfo("Registering RDD " + rdd.id + " (" + rdd.getCreationSite + ")")
    mapOutputTracker.registerShuffle(shuffleDep.shuffleId, rdd.partitions.length)
  }
  stage
}
```

根据代码清单 7-39，createShuffleMapStage 的执行步骤如下。

1）创建 ShuffleMapStage，具体如下。

① 获取 ShuffleDependency 的 rdd 属性，作为将要创建的 ShuffleMapStage 的 rdd。

② 调用 rdd（即 RDD）的 partitions 方法得到 rdd 的分区数组，此分区数组的长度即为要创建的 ShuffleMapStage 的 numTasks（Task 数量）。这说明了 map 任务数量与 RDD 的各个分区一一对应。

③ 调用 getOrCreateParentStages 方法（见代码清单 7-35）获得要创建 ShuffleMapStage 的所有父 Stage（即 parents）。

④ 生成将要创建的 ShuffleMapStage 的身份标识。

⑤ 创建 ShuffleMapStage。

2）更新刚创建的 ShuffleMapStage 的映射关系。具体如下：

① 将新创建的 ShuffleMapStage 的身份标识与 ShuffleMapStage 的映射关系放入 stageIdToStage 中。

② 将 shuffleId 与 ShuffleMapStage 的映射关系放入 shuffleIdToMapStage 中。

③ 调用 updateJobIdStageIdMaps 方法（见代码清单 7-24）更新 Job 的身份标识与 ShuffleMapStage 及其所有祖先的映射关系。

3）调用 mapOutputTracker（mapOutputTracker 这个变量名容易引起误解，其实际类型为 MapOutputTrackerMaster）的 containsShuffle 方法（见代码清单 5-70）查看是否已经存在 shuffleId 对应的 MapStatus。如果 MapOutputTrackerMaster 中未缓存 shuffleId 对应的 MapStatus，那么调用 MapOutputTrackerMaster 的 registerShuffle 方法（见代码清单 5-71）注册 shuffleId 与对应的 MapStatus 的映射关系。如果 MapOutputTrackerMaster 中缓存了 shuffleId 对应的 MapStatus，将执行以下操作：

① 调用 MapOutputTrackerMaster 的 getSerializedMapOutputStatuses（见代码清单 5-65）从 MapOutputTrackerMaster 获取对 MapStatus 序列化后的字节数组。

② 调用 MapOutputTrackerMaster 的 deserializeMapStatuses 方法对上一步获得的字节数组进行反序列化，得到 MapStatus 数组。

③ 调用 ShuffleMapStage 的 addOutputLoc 方法更新 ShuffleMapStage 的 outputLocs。

小贴示：因为 Stage 可以重试，所以当前的 Stage 可能在之前已经执行过。在上一次执行过程中，部分 map 任务可能执行成功，MapOutputTrackerMaster 中将缓存这些成功的 map 任务的 MapStatus，因而当前 Stage 只需要从 MapOutputTrackerMaster 中复制这些 MapStatus 即可，从而避免重新计算生成这些数据。

7.4.8 提交 ResultStage

handleJobSubmitted 方法（见代码清单 7-33）中处理 Job 提交的最后一步是调用 submitStage 方法（见代码清单 7-40）提交 ResultStage。

代码清单7-40 提交Stage

```
private def submitStage(stage: Stage) {
  val jobId = activeJobForStage(stage) // 获取当前Stage对应的Job的ID
  if (jobId.isDefined) {
    logDebug("submitStage(" + stage + ")")
    if (!waitingStages(stage) && !runningStages(stage) && !failedStages(stage)) {
        //当前Stage未提交
      val missing = getMissingParentStages(stage).sortBy(_.id)
      logDebug("missing: " + missing)
      if (missing.isEmpty) { // 不存在未提交的父Stage，那么提交当前Stage所有未提交的Task
        logInfo("Submitting " + stage + " (" + stage.rdd + "), which has no missing
            parents")
```

```
        submitMissingTasks(stage, jobId.get)
      } else {
        for (parent <- missing) { // 存在未提交的父Stage，那么逐个提交它们
          submitStage(parent)
        }
        waitingStages += stage
      }
    }
  } else { // 终止依赖于当前Stage的所有Job
    abortStage(stage, "No active job for stage " + stage.id, None)
  }
}
```

根据代码清单 7-40，submitStage 方法的执行步骤如下。

1）调用 activeJobForStage 方法（见代码清单 7-25）找到使用当前 Stage 的所有 ActiveJob 的身份标识。

2）如果存在 Stage 对应的 ActiveJob 的身份标识并且当前 Stage 还未提交（即 waitingStages、runningStages、failedStages 中都不包含当前 Stage），则执行以下操作：

① 调用 getMissingParentStages 方法（见代码清单 7-41）获取当前 Stage 的所有未提交的父 Stage。

② 如果不存在未提交的父 Stage，那么调用 submitMissingTasks 方法（见代码清单 7-42）提交当前 Stage 所有未提交的 Task。

③ 如果存在未提交的父 Stage，那么将多次调用 submitStage 方法提交所有未提交的父 Stage，并且将当前 Stage 加入 waitingStages 集合中（这表示当前 Stage 必须等待所有父 Stage 执行完成）。

3）如果不存在 Stage 对应的 ActiveJob 的身份标识，则调用 abortStage 方法终止依赖于当前 Stage 的所有 Job。由于 abortStage 方法的实现并不是本节最重要的内容，所以留给感兴趣的读者自行研究。

代码清单7-41　获取Stage的所有未提交的父Stage

```
private def getMissingParentStages(stage: Stage): List[Stage] = {
  val missing = new HashSet[Stage]
  val visited = new HashSet[RDD[_]]
  val waitingForVisit = new Stack[RDD[_]]
  def visit(rdd: RDD[_]) {
    if (!visited(rdd)) {
      visited += rdd
      val rddHasUncachedPartitions = getCacheLocs(rdd).contains(Nil)
      if (rddHasUncachedPartitions) {
        for (dep <- rdd.dependencies) {
          dep match {
            case shufDep: ShuffleDependency[_, _, _] =>
              val mapStage = getOrCreateShuffleMapStage(shufDep, stage.firstJobId)
              if (!mapStage.isAvailable) {
                missing += mapStage
              }
```

```
            case narrowDep: NarrowDependency[_] =>
              waitingForVisit.push(narrowDep.rdd)
          }
        }
      }
    }
    waitingForVisit.push(stage.rdd)
    while (waitingForVisit.nonEmpty) {
      visit(waitingForVisit.pop())
    }
    missing.toList
  }
```

根据代码清单 7-41，判断 Stage 的未提交父 Stage 的条件如下。

- Stage 的 RDD 的分区中存在没有对应 TaskLocation 序列的分区，即调用 getCacheLocs 方法（见代码清单 7-26）获取不到某个分区的 TaskLocation 序列，则说明当前 Stage 的某个上游 ShuffleMapStage 的某个分区任务未执行。
- Stage 的上游 ShuffleMapStage 不可用（即调用 ShuffleMapStage 的 isAvailable 方法返回 false）。

7.4.9 提交还未计算的 Task

提交 Task 的入口是 submitMissingTasks 方法（见代码清单 7-42），此方法在 Stage 没有不可用的父 Stage 时，提交当前 Stage 还未提交的任务。

代码清单7-42 提交还未计算的任务

```
private def submitMissingTasks(stage: Stage, jobId: Int) {
  stage.pendingPartitions.clear()
  // 找出当前Stage的所有分区中还没有完成计算的分区的索引
  val partitionsToCompute: Seq[Int] = stage.findMissingPartitions()
  // 获取ActiveJob的properties。properties包含了当前Job的调度、group、描述等属性信息
  val properties = jobIdToActiveJob(jobId).properties
  runningStages += stage
  stage match { // 启动对当前Stage的输出提交到HDFS的协调
    case s: ShuffleMapStage =>
      outputCommitCoordinator.stageStart(stage = s.id, maxPartitionId = s.num
          Partitions - 1)
    case s: ResultStage =>
      outputCommitCoordinator.stageStart(
        stage = s.id, maxPartitionId = s.rdd.partitions.length - 1)
  }
  val taskIdToLocations: Map[Int, Seq[TaskLocation]] = try {
    stage match { // 获取还没有完成计算的每一个分区的偏好位置
      case s: ShuffleMapStage =>
        partitionsToCompute.map { id => (id, getPreferredLocs(stage.rdd, id))}.toMap
      case s: ResultStage =>
        partitionsToCompute.map { id =>
          val p = s.partitions(id)
          (id, getPreferredLocs(stage.rdd, p))
```

```
        }.toMap
    }
  } catch { // 如果发生任何异常，则调用Stage的makeNewStageAttempt方法开始一次新的Stage执行尝试
    case NonFatal(e) =>
      stage.makeNewStageAttempt(partitionsToCompute.size)
      listenerBus.post(SparkListenerStageSubmitted(stage.latestInfo, properties))
      abortStage(stage, s"Task creation failed: $e\n${Utils.exceptionString(e)}",
          Some(e))
      runningStages -= stage
      return
  }
  // 开始Stage的执行尝试
  stage.makeNewStageAttempt(partitionsToCompute.size, taskIdToLocations.values.toSeq)
  listenerBus.post(SparkListenerStageSubmitted(stage.latestInfo, properties))

  var taskBinary: Broadcast[Array[Byte]] = null //
  try {
    val taskBinaryBytes: Array[Byte] = stage match {
      case stage: ShuffleMapStage =>
        JavaUtils.bufferToArray(
          closureSerializer.serialize((stage.rdd, stage.shuffleDep): AnyRef))
      case stage: ResultStage =>
        JavaUtils.bufferToArray(closureSerializer.serialize((stage.rdd, stage.
            func): AnyRef))
    }

    taskBinary = sc.broadcast(taskBinaryBytes) // 广播任务的序列化对象
  } catch {
    case e: NotSerializableException =>
      abortStage(stage, "Task not serializable: " + e.toString, Some(e))
      runningStages -= stage

      return
    case NonFatal(e) =>
      abortStage(stage, s"Task serialization failed: $e\n${Utils.exception-
          String(e)}", Some(e))
      runningStages -= stage
      return
  }

  val tasks: Seq[Task[_]] = try {
    stage match {
      case stage: ShuffleMapStage => // 为ShuffleMapStage的每一个分区创建一个
          ShuffleMapTask
        partitionsToCompute.map { id =>
          val locs = taskIdToLocations(id)
          val part = stage.rdd.partitions(id)
          new ShuffleMapTask(stage.id, stage.latestInfo.attemptId,
            taskBinary, part, locs, stage.latestInfo.taskMetrics, properties,
                Option(jobId),
            Option(sc.applicationId), sc.applicationAttemptId)
        }

      case stage: ResultStage => // 为ResultStage的每一个分区创建一个ResultTask
        partitionsToCompute.map { id =>
          val p: Int = stage.partitions(id)
```

```
            val part = stage.rdd.partitions(p)
            val locs = taskIdToLocations(id)
            new ResultTask(stage.id, stage.latestInfo.attemptId,
              taskBinary, part, locs, id, properties, stage.latestInfo.taskMetrics,
              Option(jobId), Option(sc.applicationId), sc.applicationAttemptId)
          }
        }
      } catch {
        case NonFatal(e) =>
          abortStage(stage, s"Task creation failed: $e\n${Utils.exceptionString(e)}",
              Some(e))
          runningStages -= stage
          return
      }

      if (tasks.size > 0) { // 调用TaskScheduler的submitTasks方法提交此批Task
        logInfo("Submitting " + tasks.size + " missing tasks from " + stage + " (" +
            stage.rdd + ")")
        stage.pendingPartitions ++= tasks.map(_.partitionId)
        logDebug("New pending partitions: " + stage.pendingPartitions)
        taskScheduler.submitTasks(new TaskSet(
          tasks.toArray, stage.id, stage.latestInfo.attemptId, jobId, properties))
        stage.latestInfo.submissionTime = Some(clock.getTimeMillis())
      } else { // 没有创建任何Task,将当前Stage标记为完成
        markStageAsFinished(stage, None)

        val debugString = stage match {
          case stage: ShuffleMapStage =>
            s"Stage ${stage} is actually done; " +
              s"(available: ${stage.isAvailable}," +
              s"available outputs: ${stage.numAvailableOutputs}," +
              s"partitions: ${stage.numPartitions})"
          case stage : ResultStage =>
            s"Stage ${stage} is actually done; (partitions: ${stage.numPartitions})"
        }
        logDebug(debugString)

        submitWaitingChildStages(stage) // 提交当前Stage的子Stage
      }
    }
```

通过对代码清单 7-42 的分析，submitMissingTasks 的执行过程总结如下。

1）清空当前 Stage 的 pendingPartitions。由于当前 Stage 的任务刚开始提交，所以需要清空便于记录需要计算的分区任务。

2）调用 Stage 的 findMissingPartitions 方法（见代码清单 7-14 和代码清单 7-16），找出当前 Stage 的所有分区中还没有完成计算的分区的索引。

3）获取 ActiveJob 的 properties。properties 包含了当前 Job 的调度、group、描述等属性信息。

4）将当前 Stage 加入 runningStages 集合中，即当前 Stage 已经处于运行状态。

5）调用 OutputCommitCoordinator 的 stageStart 方法（见代码清单 5-93），启动对当前

Stage 的输出提交到 HDFS 的协调。

6）调用 DAGScheduler 的 getPreferredLocs 方法（见代码清单 7-28），获取 partitionsToCompute 中的每一个分区的偏好位置。如果发生任何异常，则调用 Stage 的 makeNewStageAttempt 方法开始一次新的 Stage 执行尝试，然后向 listenerBus 投递 SparkListenerStageSubmitted 事件。

7）调用 Stage 的 makeNewStageAttempt 方法（见代码清单 7-13）开始 Stage 的执行尝试，并向 listenerBus 投递 SparkListenerStageSubmitted 事件。

8）如果当前 Stage 是 ShuffleMapStage，那么对 Stage 的 rdd 和 ShuffleDependency 进行序列化；如果当前 Stage 是 ResultStage，那么对 Stage 的 rdd 和对 RDD 的分区进行计算的函数 func 进行序列化。

9）调用 SparkContext 的 broadcast 方法（见代码清单 4-27）广播上一步生成的序列化对象。

10）如果当前 Stage 是 ShuffleMapStage，则为 ShuffleMapStage 的每一个分区创建一个 ShuffleMapTask。如果当前 Stage 是 ResultStage，则为 ResultStage 的每一个分区创建一个 ResultTask。

11）如果第 10）步创建了至少 1 个 Task，那么将此 Task 处理的分区索引添加到 Stage 的 pendingPartitions 中，然后为这批 Task 创建 TaskSet（即任务集合），并调用 TaskScheduler 的 submitTasks 方法提交此 TaskSet。

12）如果第 10）步没有创建任何 Task，这意味着当前 Stage 没有 Task 需要提交执行，因此调用 DAGScheduler 的 markStageAsFinished 方法（见代码清单 7-43），将当前 Stage 标记为完成。然后调用 submitWaitingChildStages 方法（见代码清单 7-44），提交当前 Stage 的子 Stage。

代码清单7-43　将Stage标记为完成

```
private def markStageAsFinished(stage: Stage, errorMessage: Option[String] =
    None): Unit = {
  val serviceTime = stage.latestInfo.submissionTime match { // 计算Stage的执行时间
    case Some(t) => "%.03f".format((clock.getTimeMillis() - t) / 1000.0)
    case _ => "Unknown"
  }
  if (errorMessage.isEmpty) {
    logInfo("%s (%s) finished in %s s".format(stage, stage.name, serviceTime))
    stage.latestInfo.completionTime = Some(clock.getTimeMillis()) // 设置Stage的完成时间

    stage.clearFailures()
  } else {
    stage.latestInfo.stageFailed(errorMessage.get) // 保存失败原因和Stage的完成时间
    logInfo(s"$stage (${stage.name}) failed in $serviceTime s due to ${error
        Message.get}")
  }

  outputCommitCoordinator.stageEnd(stage.id) // 停止对当前Stage的输出提交到HDFS的协调
```

```
    listenerBus.post(SparkListenerStageCompleted(stage.latestInfo))
    runningStages -= stage // 将当前Stage从正在运行的Stage中移除
  }
```

根据代码清单 7-43，markStageAsFinished 的执行步骤如下。

1）计算 Stage 的执行时间。

2）如果执行 Stage 的过程中没有发生错误，那么设置 Stage 的 latestInfo（即 StageInfo）的完成时间为系统当前时间，并调用 Stage 的 clearFailures 方法清空 fetchFailedAttemptIds。

3）如果执行 Stage 的过程中发生了错误，那么调用 StageInfo 的 stageFailed 方法（见代码清单 7-18）保存失败原因和 Stage 的完成时间。

4）调用 OutputCommitCoordinator 的 stageEnd 方法（见代码清单 5-94），停止对当前 Stage 的输出提交到 HDFS 的协调。

5）向 LiveListenerBus 投递 SparkListenerStageCompleted 事件。

6）将当前 Stage 从正在运行的 Stage 中移除。

代码清单7-44 提交Stage的所有处于等待中的子Stage

```
private def submitWaitingChildStages(parent: Stage) {
  // 忽略日志
  val childStages = waitingStages.filter(_.parents.contains(parent)).toArray
  waitingStages --= childStages
  for (stage <- childStages.sortBy(_.firstJobId)) {
    submitStage(stage)
  }
}
```

7.4.10 DAGScheduler 的调度流程

经过对 DAGScheduler 的详细介绍，现在我们可以用图 7-6 来表示 DAGScheduler 的主要调度流程。

这里对图 7-6 中的记号进行介绍。

记号①：表示应用程序通过对 Spark API 的调用，进行一系列 RDD 转换构建出 RDD 之间的依赖关系后，调用 DAGScheduler 的 runJob 方法将 RDD 及其血缘关系中的所有 RDD 传递给 DAGScheduler 进行调度。

记号②：DAGScheduler 的 runJob 方法实际通过调用 DAGScheduler 的 submitJob 方法向 DAGSchedulerEventProcessLoop 发送 JobSubmitted 事件。DAGSchedulerEventProcessLoop 接收到 JobSubmitted 事件后，将 JobSubmitted 事件放入事件队列（eventQueue）。

记号③：DAGSchedulerEventProcessLoop 内部的轮询线程 eventThread 不断从事件队列（eventQueue）中获取 DAGSchedulerEvent 事件，并调用 DAGSchedulerEventProcessLoop 的 doOnReceive 方法对事件进行处理。

记号④：DAGSchedulerEventProcessLoop 的 doOnReceive 方法处理 JobSubmitted 事件时，将调用 DAGScheduler 的 handleJobSubmitted 方法。handleJobSubmitted 方法将对 RDD

构建 Stage 及 Stage 之间的依赖关系。

记号⑤：DAGScheduler 首先把最上游的 Stage 中的 Task 集合提交给 TaskScheduler，然后逐步将下游的 Stage 中的 Task 集合提交给 TaskScheduler。TaskScheduler 将对 Task 集合进行调度。

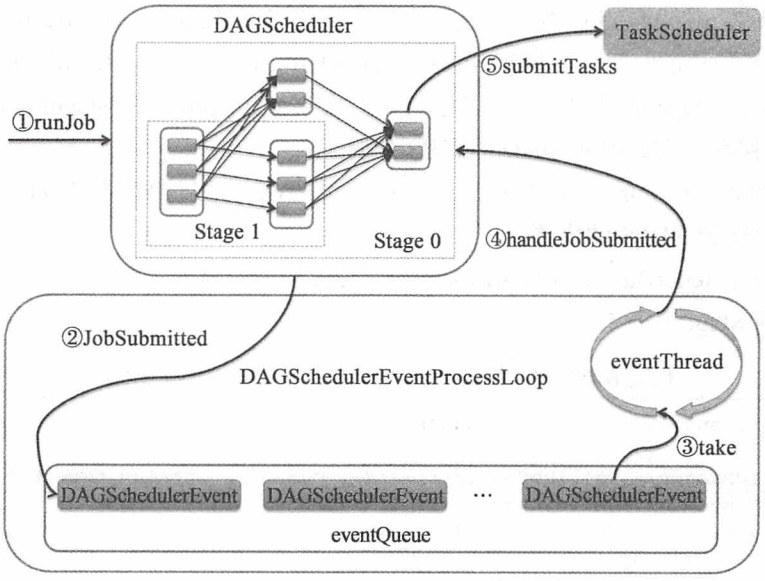


图 7-6 DAGScheduler 的调度流程

7.4.11 Task 执行结果的处理

DAGScheduler 的 taskEnded 方法（见代码清单 7-45）用于对 Task 执行的结果进行处理。对于 ShuffleMapTask 而言，需要将它的状态信息 MapStatus 追加到 ShuffleMapStage 的 outputLocs 缓存中；如果 ShuffleMapStage 的所有分区的 ShuffleMapTask 都执行成功了，那么将需要把 ShuffleMapStage 的 outputLocs 缓存中的所有 MapStatus 注册到 MapOutputTrackerMaster 的 mapStatuses 中，以便于下游 Stage 中的 Task 读取输入数据所在的位置信息；如果某个 ShuffleMapTask 执行失败了，则需要重新提交 ShuffleMapStage；如果 ShuffleMapStage 的所有 ShuffleMapTask 都执行成功了，还需要唤醒下游 Stage 的执行。对于 ResultTask 而言，如果 ResultStage 中的所有 ResultTask 都执行成功了，则将 ResultStage 标记为成功，并通知 JobWaiter 对各个 ResultTask 的执行结果进行收集，然后根据应用程序的需要进行最终的处理（如打印到控制台、输出到 HDFS）。

代码清单7-45 taskEnded方法的实现

```
def taskEnded(
    task: Task[_],
    reason: TaskEndReason,
```

```
    result: Any,
    accumUpdates: Seq[AccumulatorV2[_, _]],
    taskInfo: TaskInfo): Unit = {
  eventProcessLoop.post(
    CompletionEvent(task, reason, result, accumUpdates, taskInfo))
}
```

根据代码清单7-45，taskEnded方法将向DAGSchedulerEventProcessLoop投递CompletionEvent事件。根据代码清单7-22，DAGSchedulerEventProcessLoop接收到CompletionEvent事件后，将调用DAGScheduler的handleTaskCompletion方法。handleTaskCompletion方法中实现了针对不同执行状态的处理，这里只介绍对成功状态的处理。

1. ResultTask的结果处理

handleTaskCompletion方法对执行完成的ResultTask的处理如下。

```
val stage = stageIdToStage(task.stageId)
event.reason match {
  case Success =>
    stage.pendingPartitions -= task.partitionId
    task match {
      case rt: ResultTask[_, _] =>
        val resultStage = stage.asInstanceOf[ResultStage]
        resultStage.activeJob match {
          case Some(job) =>
            if (!job.finished(rt.outputId)) {
              updateAccumulators(event)
              job.finished(rt.outputId) = true // 对应分区的任务设置为完成状态
              job.numFinished += 1             // 将ActiveJob的已完成的任务数加一
              if (job.numFinished == job.numPartitions) {
              // ActiveJob 的所有分区的任务都完成了
                markStageAsFinished(resultStage) //将当前Stage标记为已完成
                cleanupStateForJobAndIndependentStages(job)
                listenerBus.post(
                  SparkListenerJobEnd(job.jobId, clock.getTimeMillis(), Job
                      Succeeded))
              }

              try {
                // 由JobWaiter的resultHandler函数来处理Job中每个Task的执行结果
                job.listener.taskSucceeded(rt.outputId, event.result)
              } catch {
                case e: Exception =>
                  job.listener.jobFailed(new SparkDriverExecutionException(e))//
                      由JobWaiter处理失败
              }
            }
          case None =>
            logInfo("Ignoring result from " + rt + " because its job has
                finished")
        }

      case smt: ShuffleMapTask =>
        // 忽略ShuffleMapTask的结果处理
```

根据上述代码，对 ResultTask 的执行结果进行处理的步骤如下。

1）将 ResultStage 的 ActiveJob 的 finished 里对应分区的任务设置为完成状态，并且将 ActiveJob 的已完成的任务数（numFinished）加 1。

2）如果 ActiveJob 的所有任务都完成，则调用 DAGScheduler 的 markStageAsFinished 方法（见代码清单 7-43）标记当前 Stage 完成，并向 LiveListenerBus 投递 SparkListener-JobEnd 事件。

3）调用 JobWaiter 的 taskSucceeded 方法（见代码清单 7-20），进而调用 JobWaiter 的 resultHandler 函数来处理 Job 中每个 Task 的执行结果。

2. ShuffleMapTask 的结果处理

handleTaskCompletion 方法对执行完成的 ShuffleMapTask 的处理如下。

```
val stage = stageIdToStage(task.stageId)
event.reason match {
  case Success =>
    stage.pendingPartitions -= task.partitionId
    task match {
      case rt: ResultTask[_, _] =>
        // 忽略ResultTask的结果处理
      case smt: ShuffleMapTask =>
        val shuffleStage = stage.asInstanceOf[ShuffleMapStage]
        updateAccumulators(event)
        val status = event.result.asInstanceOf[MapStatus]
        val execId = status.location.executorId
        logDebug("ShuffleMapTask finished on " + execId)
        if (failedEpoch.contains(execId) && smt.epoch <= failedEpoch(execId)) {
          logInfo(s"Ignoring possibly bogus $smt completion from executor
              $execId")
        } else { //将分区任务的MapStatus追加到Stage的outputLocs中
          shuffleStage.addOutputLoc(smt.partitionId, status)
        }

        if (runningStages.contains(shuffleStage) && shuffleStage.pendingPartitions.
            isEmpty) {
          markStageAsFinished(shuffleStage) // 将ShuffleMapStage标记为完成
          logInfo("looking for newly runnable stages")
          logInfo("running: " + runningStages)
          logInfo("waiting: " + waitingStages)
          logInfo("failed: " + failedStages)
          // 将当前Stage的shuffleId和outputLocs中的MapStatus注册到MapOutput-TrackerMaster的
          // mapStatuses中
          mapOutputTracker.registerMapOutputs(
            shuffleStage.shuffleDep.shuffleId,
            shuffleStage.outputLocInMapOutputTrackerFormat(),
            changeEpoch = true)

          clearCacheLocs()

          if (!shuffleStage.isAvailable) {
            logInfo("Resubmitting " + shuffleStage + " (" + shuffleStage.name +
              ") because some of its tasks had failed: " +
```

```
            shuffleStage.findMissingPartitions().mkString(", "))
          submitStage(shuffleStage) // 有任务失败了，这时需要再次提交此ShuffleMapStage
        } else {
          if (shuffleStage.mapStageJobs.nonEmpty) {
            val stats = mapOutputTracker.getStatistics(shuffleStage.shuffleDep)
            for (job <- shuffleStage.mapStageJobs) {
              markMapStageJobAsFinished(job, stats) // 将ActiveJob标记为执行成功
            }
          }
          submitWaitingChildStages(shuffleStage) // 提交当前ShuffleMapStage的下游Stage
        }
      }
    }
```

根据上述代码，对 ShuffleMapTask 的执行结果进行处理的步骤如下。

1）将 Task 的 partitionId 和 MapStatus 追加到 Stage 的 outputLocs 中。

2）如果 ShuffleMapStage 中没有待计算的分区，那么调用 DAGScheduler 的 markStageAsFinished 方法（见代码清单 7-43），将 ShuffleMapStage 标记为完成，然后调用 MapOutputTrackerMaster 的 registerMapOutputs 方法（见代码清单 5-72），将当前 Stage 的 shuffleId 和 outputLocs 中的 MapStatus 注册到 MapOutputTrackerMaster 的 mapStatuses 中。根据 5.6 节的内容，这里注册的 MapStatus 将最终被 reduce 任务所用。

3）如果 ShuffleMapStage 的 isAvailable 方法返回 false（即 _numAvailableOutputs 与 numPartitions 不相等），那么说明有任务失败了，这时需要再次提交此 ShuffleMapStage。

4）如果 ShuffleMapStage 的 isAvailable 方法返回 true（即 _numAvailableOutputs 与 numPartitions 相等），那么说明所有任务执行成功了，这时调用 MapOutputTracker 的 getStatistics 方法获取 Shuffle 依赖的各个 map 任务输出 Block 大小的统计信息，并将 ShuffleMapStage 的 _mapStageJobs 属性中保存的各个 ActiveJob 标记为执行成功，最后调用 submitWaitingChildStages 方法（见代码清单 7-44），提交当前 ShuffleMapStage 的子 Stage。

通过本节对 DAGScheduler 的分析，可以看到 DAGScheduler 的主要工作包括创建 Job，构建 Stage 的上下游关系，反向驱动 Stage 的提交及正向提交 Stage 等。反向驱动 Stage 和正向提交 Stage 的过程可以用图 7-7 来表示。

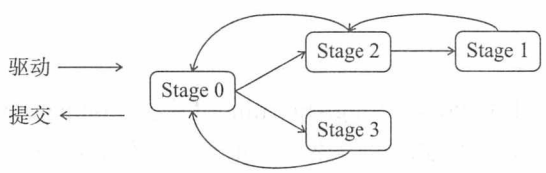


图 7-7 Stage 的正向驱动与反向提交

7.5 调度池 Pool

TaskSchedulerImpl 对 Task 的调度依赖于调度池 Pool，所有需要被调度的 TaskSet 都被

置于调度池中。调度池 Pool 通过调度算法对每个 TaskSet 进行调度，并将被调度的 TaskSet 交给 TaskSchedulerImpl 进行资源调度。

7.5.1 调度算法

调度池对 TaskSet 的调度取决于调度算法，所以本节将介绍调度算法。特质 SchedulingAlgorithm 定义了调度算法的规范，代码如下。

```
private[spark] trait SchedulingAlgorithm {
  def comparator(s1: Schedulable, s2: Schedulable): Boolean
}
```

根据上面的代码，我们知道 SchedulingAlgorithm 仅仅定义了一个 comparator 方法，用于对两个 Schedulable 进行比较。我们暂时不用去了解 Schedulable，先来看看调度算法的具体实现。

SchedulingAlgorithm 有两个实现类，分别为实现了先进先出（First In First Out，FIFO）算法的 FIFOSchedulingAlgorithm 和公平调度算法的 FairSchedulingAlgorithm，下面将对这两种算法实现进行介绍。

1. FIFOSchedulingAlgorithm 详解

FIFOSchedulingAlgorithm 实现了 FIFO 调度算法，如代码清单 7-46 所示。

代码清单7-46 FIFOSchedulingAlgorithm的实现

```
private[spark] class FIFOSchedulingAlgorithm extends SchedulingAlgorithm {
  override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
    val priority1 = s1.priority
    val priority2 = s2.priority
    var res = math.signum(priority1 - priority2) // 对s1和s2两个Schedulable的优先级
        进行比较
    if (res == 0) {
      val stageId1 = s1.stageId
      val stageId2 = s2.stageId
      res = math.signum(stageId1 - stageId2) // 对s1和s2所属的Stage的身份标识进行比较
    }
    res < 0
  }
}
```

根据代码清单 7-46，FIFOSchedulingAlgorithm 重写的 comparator 方法的执行步骤如下。

1）对 s1 和 s2 两个 Schedulable 的优先级（值越小，优先级越高）进行比较。

2）如果两个 Schedulable 的优先级相同，则对 s1 和 s2 所属的 Stage 的身份标识进行比较。

3）如果比较的结果小于 0，则优先调度 s1，否则优先调度 s2。

2. FairSchedulingAlgorithm 详解

FairSchedulingAlgorithm 实现了公平调度算法，如代码清单 7-47 所示。

代码清单7-47 FairSchedulingAlgorithm的实现

```
private[spark] class FairSchedulingAlgorithm extends SchedulingAlgorithm {
  override def comparator(s1: Schedulable, s2: Schedulable): Boolean = {
    val minShare1 = s1.minShare
    val minShare2 = s2.minShare
    val runningTasks1 = s1.runningTasks
    val runningTasks2 = s2.runningTasks
    val s1Needy = runningTasks1 < minShare1
    val s2Needy = runningTasks2 < minShare2
    val minShareRatio1 = runningTasks1.toDouble / math.max(minShare1, 1.0)
    val minShareRatio2 = runningTasks2.toDouble / math.max(minShare2, 1.0)
    val taskToWeightRatio1 = runningTasks1.toDouble / s1.weight.toDouble
    val taskToWeightRatio2 = runningTasks2.toDouble / s2.weight.toDouble

    var compare = 0
    if (s1Needy && !s2Needy) {
      return true
    } else if (!s1Needy && s2Needy) {
      return false
    } else if (s1Needy && s2Needy) {
      compare = minShareRatio1.compareTo(minShareRatio2)
    } else {
      compare = taskToWeightRatio1.compareTo(taskToWeightRatio2)
    }
    if (compare < 0) {
      true
    } else if (compare > 0) {
      false
    } else {
      s1.name < s2.name
    }
  }
```

根据代码清单 7-47，FairSchedulingAlgorithm 重写的 comparator 方法的执行步骤如下。

1）如果 s1 中处于运行状态的 Task 的数量小于 s1 的 minShare，并且 s2 中处于运行状态的 Task 的数量大于等于 s2 的 minShare，那么优先调度 s1。

2）如果 s1 中处于运行状态的 Task 的数量大于等于 s1 的 minShare，并且 s2 中处于运行状态的 Task 的数量小于 s2 的 minShare，那么优先调度 s2。

3）如果 s1 中处于运行状态的 Task 的数量小于 s1 的 minShare，并且 s2 中处于运行状态的 Task 的数量小于 s2 的 minShare，那么再对 minShareRatio1 和 minShareRatio2 进行比较。如果 minShareRatio1 小于 minShareRatio2，则优先调度 s1；如果 minShareRatio2 小于 minShareRatio1，则优先调度 s2。如果 minShareRatio1 和 minShareRatio2 相等，还需要对 s1 和 s2 的名字进行比较。如果 s1 的名字小于 s2 的名字，则优先调度 s1，否则优先调度 s2。minShareRatio 是正在运行的任务数量与 minShare 之间的比值。

4）如果 s1 中处于运行状态的 Task 的数量大于等于 s1 的 minShare，并且 s2 中处于运行状态的 Task 的数量大于等于 s2 的 minShare，那么再对 taskToWeightRatio1 和 taskToWeightRatio2 进行比较。如果 taskToWeightRatio1 小于 taskToWeightRatio2，则优先调度

s1；如果 taskToWeightRatio2 小于 taskToWeightRatio1，则优先调度 s2。如果 taskToWeightRatio1 和 taskToWeightRatio2 相等，还需要对 s1 和 s2 的名字进行比较。如果 s1 的名字小于 s2 的名字，则优先调度 s1，否则优先调度 s2。taskToWeightRatio 是正在运行的任务数量与权重（weight）之间的比值。

7.5.2 Pool 的实现

TaskScheduler 对任务的调度是借助于调度池实现的，Pool 是对 Task 集合进行调度的调度池。调度池内部有一个根调度队列，根调度队列中包含了多个子调度池。子调度池自身的调度队列中还可以包含其他的调度池或者 TaskSetManager（TaskSetManager 将在 7.6 节详细介绍，读者这里只需知道每个 TaskSetManager 包含着 Task 的集合），所以整个调度池是一个多层次的调度队列。Pool 实现了 Schedulable 特质，其中包含如下属性。

- parent：当前 Pool 的父 Pool。
- poolName：Pool 的构造器属性之一，表示 Pool 的名称。
- schedulingMode：Pool 的构造器属性之一，表示调度模式（SchedulingMode）。枚举类型 SchedulingMode 共有 FAIR、FIFO、NONE 三种枚举值。
- initMinShare：minShare 的初始值。
- initWeight：weight 的初始值。
- weight：用于公平调度算法的权重。
- minShare：用于公平调度算法的参考值。
- schedulableQueue：类型为 ConcurrentLinkedQueue[Schedulable]，用于存储 Schedulable。由于 Schedulable 只有 Pool 和 TaskSetManager 两个实现类，所以我们知道 schedulableQueue 是一个可以嵌套的层次结构，如图 7-8 所示。

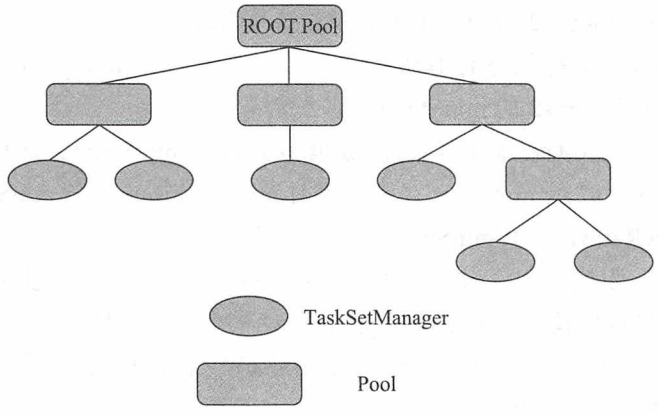


图 7-8 调度队列的层次关系

小贴士：熟悉 Hadoop YARN 的读者会发现，Spark 的调度池中的调度队列与 YARN 中调度队列的设计非常相似，也采用了层级队列的管理方式。

- schedulableNameToSchedulable：调度名称与 Schedulable 的对应关系。
- runningTasks：当前正在运行的任务数量。
- priority：进行调度的优先级。
- stageId：调度池或 TaskSetManager 所属 Stage 的身份标识。
- name：与 poolName 相同。
- taskSetSchedulingAlgorithm：任务集合的调度算法，默认为 FIFOSchedulingAlgorithm。

了解了 Pool 的属性，现在来看看 Pool 中提供的方法。

1. addSchedulable

addSchedulable 方法（见代码清单 7-48）用于将 Schedulable 添加到 schedulableQueue 和 schedulableNameToSchedulable 中，并将 Schedulable 的父亲设置为当前 Pool。

代码清单7-48 添加Schedulable

```
override def addSchedulable(schedulable: Schedulable) {
  require(schedulable != null)
  schedulableQueue.add(schedulable)
  schedulableNameToSchedulable.put(schedulable.name, schedulable)
  schedulable.parent = this
}
```

2. removeSchedulable

removeSchedulable 方法（见代码清单 7-49）用于将指定的 Schedulable 从 schedulableQueue 和 schedulableNameToSchedulable 中移除。

代码清单7-49 移除Schedulable

```
override def removeSchedulable(schedulable: Schedulable) {
  schedulableQueue.remove(schedulable)
  schedulableNameToSchedulable.remove(schedulable.name)
}
```

3. getSchedulableByName

getSchedulableByName 方法（见代码清单 7-50）用于根据指定名称查找 Schedulable。

代码清单7-50 根据指定名称查找Schedulable

```
override def getSchedulableByName(schedulableName: String): Schedulable = {
  if (schedulableNameToSchedulable.containsKey(schedulableName)) {
    return schedulableNameToSchedulable.get(schedulableName) // 从当前Pool找到的指
        定名称的Schedulable
  }
  for (schedulable <- schedulableQueue.asScala) {
    val sched = schedulable.getSchedulableByName(schedulableName) //从子Schedulable
        中查找
```

```
    if (sched != null) {
      return sched
    }
  }
  null
}
```

根据代码清单 7-50，getSchedulableByName 方法的执行步骤如下。

1）从当前 Pool 的 schedulableNameToSchedulable 中查找指定名称的 Schedulable。

2）如果上一步未找到，则从当前 Pool 的 schedulableQueue 中的各个子 Schedulable 中查找，直到找到指定名称的 Schedulable 或者返回 null。

4. executorLost

executorLost 方法（见代码清单 7-51）用于当某个 Executor 丢失后，调用当前 Pool 的 schedulableQueue 中的各个 Schedulable（可能为子调度池，也可能是 TaskSetManager）的 executorLost 方法。TaskSetManager 的 executorLost 方法（见代码清单 7-74）进而将在此 Executor 上正在运行的 Task 作为失败任务处理，并重新提交这些任务。

代码清单7-51　executorLost的实现

```
override def executorLost(executorId: String, host: String, reason: Executor
    LossReason) {
  schedulableQueue.asScala.foreach(_.executorLost(executorId, host, reason))
}
```

5. checkSpeculatableTasks

checkSpeculatableTasks 方法（见代码清单 7-52）用于检查当前 Pool 中是否有需要推断执行的任务。checkSpeculatableTasks 实际通过迭代调用 schedulableQueue 中的各个子 Schedulable 的 checkSpeculatableTasks 方法来实现。Pool 的 checkSpeculatableTasks 方法和 TaskSetManager 的 checkSpeculatableTasks 方法（见代码清单 7-62），一起实现了按照深度遍历算法从调度池中查找可推断执行的任务。

代码清单7-52　检查当前Pool中是否有需要推断执行的任务

```
override def checkSpeculatableTasks(minTimeToSpeculation: Int): Boolean = {
  var shouldRevive = false
  for (schedulable <- schedulableQueue.asScala) {
    shouldRevive |= schedulable.checkSpeculatableTasks(minTimeToSpeculation)
  }
  shouldRevive
}
```

6. getSortedTaskSetQueue

getSortedTaskSetQueue 方法（见代码清单 7-53）用于对当前 Pool 中的所有 TaskSetManager 按照调度算法进行排序，并返回排序后的 TaskSetManager。getSortedTaskSet-Queue 实际是通过迭代调用 schedulableQueue 中的各个子 Schedulable 的 getSortedTaskSetQueue 方法来实现。

代码清单7-53 getSortedTaskSetQueue的实现

```
override def getSortedTaskSetQueue: ArrayBuffer[TaskSetManager] = {
  var sortedTaskSetQueue = new ArrayBuffer[TaskSetManager]
  val sortedSchedulableQueue =
    schedulableQueue.asScala.toSeq.sortWith(taskSetSchedulingAlgorithm.comparator)
  for (schedulable <- sortedSchedulableQueue) {
    sortedTaskSetQueue ++= schedulable.getSortedTaskSetQueue
  }
  sortedTaskSetQueue
}
```

7. increaseRunningTasks

increaseRunningTasks 方法（见代码清单 7-54）用于增加当前 Pool 及其父 Pool 中记录的当前正在运行的任务数量。

代码清单7-54 增加正在运行的Task数量

```
def increaseRunningTasks(taskNum: Int) {
  runningTasks += taskNum
  if (parent != null) {
    parent.increaseRunningTasks(taskNum)
  }
}
```

8. decreaseRunningTasks

decreaseRunningTasks 方法（见代码清单 7-55）用于减少当前 Pool 及其父 Pool 中记录的当前正在运行的任务数量。

代码清单7-55 减少正在运行的Task数量

```
def decreaseRunningTasks(taskNum: Int) {
  runningTasks -= taskNum
  if (parent != null) {
    parent.decreaseRunningTasks(taskNum)
  }
}
```

7.5.3 调度池构建器

在创建了调度池之后，池内还没有“水”，更没有“鱼”。需要园林建造师向池中注入水，有了水之后就可以养育多种多样的鱼儿了。SchedulableBuilder 正是构建调度池的“建造师”。

特质 SchedulableBuilder 定义了调度池构建器的行为规范，请看如下代码。

```
private[spark] trait SchedulableBuilder {
  def rootPool: Pool
  def buildPools(): Unit
  def addTaskSetManager(manager: Schedulable, properties: Properties): Unit
}
```

上述代码中一共定义了三个方法。

- rootPool：返回根调度池。
- buildPools：对调度池进行构建。
- addTaskSetManager：向调度池内添加 TaskSetManager。

针对 FIFO 和 Fair 两种调度算法，SchedulableBuilder 共有两种实现，分别是 FIFOSchedulableBuilder 和 FairSchedulableBuilder。

1. FIFOSchedulableBuilder 详解

FIFOSchedulableBuilder 的实现如代码清单 7-56 所示。

代码清单7-56 FIFOSchedulableBuilder的实现

```
private[spark] class FIFOSchedulableBuilder(val rootPool: Pool)
  extends SchedulableBuilder with Logging {
  override def buildPools() {
    // nothing
  }

  override def addTaskSetManager(manager: Schedulable, properties: Properties) {
    rootPool.addSchedulable(manager)
  }
}
```

根据代码清单 7-56，FIFOSchedulableBuilder 实现的 buildPools 方法是个空方法，而实现的 addTaskSetManager 方法将向根调度池中添加 TaskSetManager。

有了对 FIFOSchedulableBuilder 的实现分析，我们可以用图 7-9 来表示由 FIFOSchedulableBuilder 构建出的调度池的内存结构。

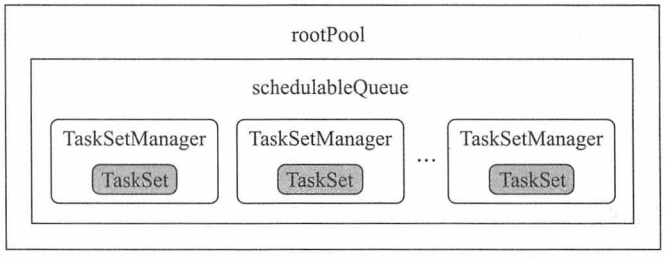


图 7-9 FIFOSchedulableBuilder 构建的调度池的内存结构

2. FairSchedulableBuilder 详解

FairSchedulableBuilder 的实现较为复杂，为便于分析，我们从了解它的属性开始。FairSchedulableBuilder 的属性如下。

1）rootPool：根调度池。rootPool 是 FairSchedulableBuilder 的构造器属性。

2）conf：即 SparkConf。

3）schedulerAllocFile：用户指定的文件系统中的调度分配文件。此文件可以通过 spark.

scheduler.allocation.file 属性配置，FairSchedulableBuilder 将从文件系统中读取此文件提供的公平调度配置。

4）DEFAULT_SCHEDULER_FILE：默认的调度文件名。常量 DEFAULT_SCHEDULER_FILE 的值固定为 "fairscheduler.xml"，FairSchedulableBuilder 将从 ClassPath 中读取此文件提供的公平调度配置。下面展示了 fairscheduler.xml 的配置内容。

```
<allocations>
  <pool name="production">
    <schedulingMode>FAIR</schedulingMode>
    <weight>1</weight>
    <minShare>2</minShare>
  </pool>
  <pool name="test">
    <schedulingMode>FIFO</schedulingMode>
    <weight>2</weight>
    <minShare>3</minShare>
  </pool>
</allocations>
```

从 fairscheduler.xml 的配置内容可以看到，FAIR 调度算法下的每个调度池可以选择不同的调度算法。

① FAIR_SCHEDULER_PROPERTIES：即 spark.scheduler.pool，此属性的值作为放置 TaskSetManager 的公平调度池的名称。

② DEFAULT_POOL_NAME：默认的调度池名。常量 DEFAULT_POOL_NAME 的值固定为 "default"。

③ MINIMUM_SHARES_PROPERTY：常量 MINIMUM_SHARES_PROPERTY 的值固定为 "minShare"，即 XML 文件的 <Pool> 节点的子节点 <mindshare>。节点 <mindshare> 的值将作为 Pool 的 minShare 属性。

④ SCHEDULING_MODE_PROPERTY：常量 SCHEDULING_MODE_PROPERTY 的值固定为 "schedulingMode"，即 XML 文件的 <Pool> 节点的子节点 <schedulingMode>。节点 <schedulingMode> 的值将作为 Pool 的调度模式（schedulingMode）属性。

⑤ WEIGHT_PROPERTY：权重属性。常量 WEIGHT_PROPERTY 的值固定为 "weight"，即 XML 文件的 <Pool> 节点的子节点 <weight>。节点 <weight> 的值将作为 Pool 的权重（weight）属性。

⑥ POOL_NAME_PROPERTY：常量 POOL_NAME_PROPERTY 的值固定为 "@name"，即 XML 文件的 <Pool> 节点的 name 属性。name 属性的值将作为 Pool 的调度池名（poolName）属性。

⑦ POOLS_PROPERTY：调度池属性。常量 POOLS_PROPERTY 的值固定为 "pool"。

⑧ DEFAULT_SCHEDULING_MODE：默认的调度模式 FIFO。

⑨ DEFAULT_MINIMUM_SHARE：公平调度算法中 Schedulable 的 minShare 属性的默认值。常量 DEFAULT_MINIMUM_SHARE 的值固定为 0。

书籍是人类进步的阶梯

⑩ DEFAULT_WEIGHT：默认的权重。常量 DEFAULT_WEIGHT 的值固定为 1。

有了对 FairSchedulableBuilder 属性的了解，现在来看看 FairSchedulableBuilder 的实现。FairSchedulableBuilder 实现了特质 SchedulableBuilder 的 buildPools 和 addTaskSet-Manager，下面将逐一介绍。

5）buildPools：FairSchedulableBuilder 实现的 buildPools 方法如代码清单 7-57 所示。

代码清单7-57 构建公平调度池

```
override def buildPools() {
  var is: Option[InputStream] = None
  try {
    is = Option {
      schedulerAllocFile.map { f =>
        new FileInputStream(f) // 从文件系统中读取公平调度配置的文件输入流
      }.getOrElse {
        Utils.getSparkClassLoader.getResourceAsStream(DEFAULT_SCHEDULER_FILE)
      }
    }

    is.foreach { i => buildFairSchedulerPool(i) } // 解析文件输入流并构建调度池
  } finally {
    is.foreach(_.close())
  }

  buildDefaultPool() // 构建默认的调度池
}
```

根据代码清单 7-57，FairSchedulableBuilder 的 buildPools 方法的执行步骤如下。

①优先从文件系统中读取 schedulerAllocFile 的文件输入流。如果用户未指定此 scheduler-AllocFile，则默认从类路径下加载 fairscheduler.xml 文件的输入流。

②调用 buildFairSchedulerPool 方法（见代码清单 7-58）对文件输入流进行解析并构建调度池。

③调用 buildDefaultPool 方法（见代码清单 7-59）构建默认的调度池。

代码清单7-58 buildFairSchedulerPool的实现

```
private def buildFairSchedulerPool(is: InputStream) {
  val xml = XML.load(is) // 将文件输入流转换为XML
  for (poolNode <- (xml \\ POOLS_PROPERTY)) { // 读取XML的每一个<Pool>节点

    val poolName = (poolNode \ POOL_NAME_PROPERTY).text // 读取<Pool>的name属性作为
        调度池的名称
    var schedulingMode = DEFAULT_SCHEDULING_MODE
    var minShare = DEFAULT_MINIMUM_SHARE
    var weight = DEFAULT_WEIGHT

    val xmlSchedulingMode = (poolNode \ SCHEDULING_MODE_PROPERTY).text
    if (xmlSchedulingMode != "") {
      try { // 读取<Pool>的子节点<schedulingMode>的值作为调度池的调度模式（schedulingMode）
            属性
```

书籍是人类进步的阶梯

```
        schedulingMode = SchedulingMode.withName(xmlSchedulingMode)
      } catch {
        case e: NoSuchElementException =>
          logWarning(s"Unsupported schedulingMode: $xmlSchedulingMode, " +
            s"using the default schedulingMode: $schedulingMode")
      }
    }
    // 读取<Pool>的子节点<minShare>的值作为调度池的minShare属性
    val xmlMinShare = (poolNode \ MINIMUM_SHARES_PROPERTY).text
    if (xmlMinShare != "") {
      minShare = xmlMinShare.toInt
    }
    // 读取<Pool>的子节点<weight>的值作为调度池的权重（weight）属性
    val xmlWeight = (poolNode \ WEIGHT_PROPERTY).text
    if (xmlWeight != "") {
      weight = xmlWeight.toInt
    }
    val pool = new Pool(poolName, schedulingMode, minShare, weight) // 创建子调度池
    rootPool.addSchedulable(pool) // 将创建的子调度池添加到根调度池的调度队列
    logInfo("Created pool %s, schedulingMode: %s, minShare: %d, weight: %d".format(
      poolName, schedulingMode, minShare, weight))
  }
}
```

根据代码清单 7-58，buildFairSchedulerPool 方法的执行步骤如下。

① 将文件输入流转换为 XML。

② 读取 XML 的每一个 <Pool> 节点，并对每个 <Pool> 节点的子元素执行以下读取操作。

- 读取 <Pool> 的 name 属性作为调度池的名称。
- 读取 <Pool> 的子节点 <schedulingMode> 的值作为调度池的调度模式（schedulingMode）属性。如果未对 <Pool> 节点指定子节点 <schedulingMode>，则默认以常量 DEFAULT_SCHEDULING_MODE 的值作为调度池的调度模式。
- 读取 <Pool> 的子节点 <minShare> 的值作为调度池的 minShare 属性。如果未对 <Pool> 节点指定子节点 <minShare>，则默认以常量 DEFAULT_MINIMUM_SHARE 的值作为调度池的 minShare 属性。
- 读取 <Pool> 的子节点 <weight> 的值作为调度池的权重（weight）属性。如果未对 <Pool> 节点指定子节点 <weight>，则默认以常量 DEFAULT_WEIGHT 的值作为调度池的 weight 属性。
- 使用调度池名称、schedulingMode、minShare 及 weight 创建子调度池。
- 将创建的子调度池添加到根调度池的调度队列中。

构建默认调度池的代码如代码清单 7-59 所示。

代码清单7-59 构建默认调度池

```
private def buildDefaultPool() {
  if (rootPool.getSchedulableByName(DEFAULT_POOL_NAME) == null) {
    val pool = new Pool(DEFAULT_POOL_NAME, DEFAULT_SCHEDULING_MODE,
```

书籍是人类进步的阶梯

```
        DEFAULT_MINIMUM_SHARE, DEFAULT_WEIGHT) // 创建默认调度池
      rootPool.addSchedulable(pool)              // 向根调度池的调度队列中添加默认的子调度池
      logInfo("Created default pool %s, schedulingMode: %s, minShare: %d, weight:
          %d".format(
        DEFAULT_POOL_NAME, DEFAULT_SCHEDULING_MODE, DEFAULT_MINIMUM_SHARE, DEFAULT_
            WEIGHT))
    }
  }
```

根据代码清单 7-59，buildDefaultPool 方法用于当根调度池及其子调度池中不存在名为 default 的调度池时，执行如下操作。

① 创建名为 default，schedulingMode 为 FIFO，minShare 为 0，weight 为 1 的默认子调度池。

② 向根调度池的调度队列中添加默认的子调度池。

6）addTaskSetManager：FairSchedulableBuilder 实现的 addTaskSetManager 方法如代码清单 7-60 所示。

代码清单7-60 添加TaskSetManager

```
override def addTaskSetManager(manager: Schedulable, properties: Properties) {
  var poolName = DEFAULT_POOL_NAME
  var parentPool = rootPool.getSchedulableByName(poolName) //以默认调度池作为
      TaskSetManager的父调度池
  if (properties != null) { // 以spark.scheduler.pool属性指定的调度池作为TaskSetManager
      的父调度池
    poolName = properties.getProperty(FAIR_SCHEDULER_PROPERTIES, DEFAULT_POOL_NAME)
    parentPool = rootPool.getSchedulableByName(poolName)
    if (parentPool == null) {
      parentPool = new Pool(poolName, DEFAULT_SCHEDULING_MODE,
        DEFAULT_MINIMUM_SHARE, DEFAULT_WEIGHT)
      rootPool.addSchedulable(parentPool)
      logInfo("Created pool %s, schedulingMode: %s, minShare: %d, weight: %d".format(
        poolName, DEFAULT_SCHEDULING_MODE, DEFAULT_MINIMUM_SHARE, DEFAULT_WEIGHT))
    }
  }
  parentPool.addSchedulable(manager) // 将TaskSetManager放入父调度池
  logInfo("Added task set " + manager.name + " tasks to pool " + poolName)
}
```

根据代码清单 7-60，FairSchedulableBuilder 的 addTaskSetManager 方法的执行步骤如下。

① 以名为 default 的调度池作为 TaskSetManager 的父调度池。

② 如果在 Properties 中指定了 spark.scheduler.pool 属性，则从根调度池中查找此属性值对应的调度池作为 TaskSetManager 的父调度池。如果根调度池中不存在此属性值对应的调度池，则创建以此属性值为名称的调度池作为 TaskSetManager 的父调度池，并将此调度池作为根调度池的子调度池。

③ 将 TaskSetManager 放入前两步确定的父调度池。

书籍是人类进步的阶梯

有了对 FairSchedulableBuilder 的实现分析，我们可以用图 7-10 来表示由 FairSchedulable-Builder 构建出的调度池的内存结构。

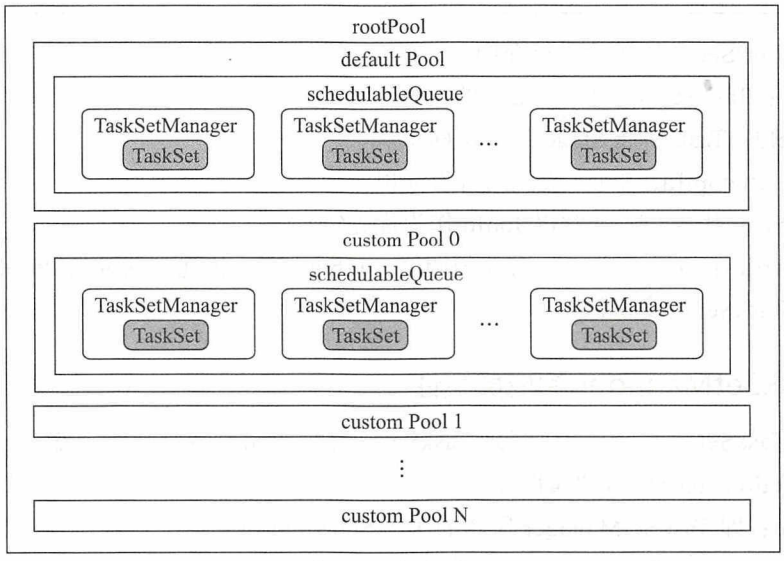


图 7-10 FairSchedulableBuilder 构建的调度池的内存结构

图 7-10 主要展示了根调度池中的多个子调度池，子调度池中包含默认的子调度池。每个调度池中都有若干 TaskSetManager。

7.6 任务集合管理器 TaskSetManager

TaskSetManager 也实现了 Schedulable 特质，并参与到调度池的调度中。TaskSetManager 对 TaskSet 进行管理，包括任务推断、Task 本地性，并对 Task 进行资源分配。TaskScheduler-Impl 依赖于 TaskSetManager，本节将对 TaskSetManager 的实现进行分析。

7.6.1 Task 集合

根据之前的介绍，DAGScheduler 将 Task 提交给 TaskScheduler 时，需要将多个 Task 打包为 TaskSet。TaskSet 是整个调度池中对 Task 进行调度管理的基本单位，由调度池中的 TaskSetManager 来管理，它的定义如代码清单 7-61 所示。

代码清单7-61 TaskSet的定义

```
private[spark] class TaskSet(
    val tasks: Array[Task[_]],
    val stageId: Int,
    val stageAttemptId: Int,
    val priority: Int,
```

书籍是人类进步的阶梯

```
    val properties: Properties) {
  val id: String = stageId + "." + stageAttemptId
  override def toString: String = "TaskSet " + id
}
```

这里对 TaskSet 中的属性进行介绍。

- tasks：TaskSet 所包含的 Task 的数组。
- stageId：Task 所属 Stage 的身份标识。
- stageAttemptId：Stage 尝试的身份标识。
- priority：优先级。通常以 JobId 作为优先级。
- properties：包含了与 Job 有关的调度、Job group、描述等属性的 Properties。
- id：TaskSet 的身份标识。

7.6.2 TaskSetManager 的成员属性

要了解 TaskSetManager 如何管理 TaskSet，对它的属性成员进行了解，有助于读者的理解，TaskSetManager 包含的属性如下。

- sched：即 TaskSetManager 所属的 TaskSchedulerImpl。
- taskSet：当前 TaskSetManager 管理的 TaskSet。
- maxTaskFailures：最大任务失败次数。
- clock：系统时钟。
- conf：即 SparkConf。
- SPECULATION_QUANTILE：开始推断执行的任务分数。可以通过 spark.speculation.quantile 属性进行配置，默认为 0.75。
- SPECULATION_MULTIPLIER：推断的乘数。可通过 spark.speculation.multiplier 属性进行配置，默认为 1.5。SPECULATION_MULTIPLIER 将用于可推断执行 Task 的检查。
- maxResultSize：结果总大小的字节限制。可通过 spark.driver.maxResultSize 属性进行配置，默认为 1GB。maxResultSize 是通过调用 Utils 工具类的 getMaxResultSize 方法获得的，具体可参阅附录 A。
- env：即 SparkEnv。
- ser：即 SparkEnv 的 closureSerializer 的实例。
- tasks：TaskSet 包含的 Task 数组，即 TaskSet 的 tasks 属性。
- numTasks：TaskSet 包含的 Task 的数量，即 tasks 数组的长度。
- copiesRunning：对每个 Task 的复制运行数进行记录的数组。copiesRunning 按照索引与 tasks 数组的同一索引位置的 Task 相对应，记录对应 Task 的复制运行数量。
- successful：对每个 Task 是否执行成功进行记录的数组。successful 按照索引与 tasks 数组的同一索引位置的 Task 相对应，记录对应的 Task 是否执行成功。

书籍是人类进步的阶梯

- numFailures：对每个 Task 的执行失败次数进行记录的数组。numFailures 按照索引与 tasks 数组的同一索引位置的 Task 相对应，记录对应 Task 的执行失败次数。
- taskAttempts：对每个 Task 的所有执行尝试信息进行记录的数组。taskAttempts 按照索引与 tasks 数组的同一索引位置的 Task 相对应，记录对应 Task 的所有 Task 尝试信息（即 TaskInfo，读者暂时只需要知道这个对象即可）。
- tasksSuccessful：执行成功的 Task 数量。
- weight：用于公平调度算法的权重。
- minShare：用于公平调度算法的参考值。
- priority：进行调度的优先级。
- stageId：调度池所属的 Stage 的身份标识。
- name：TaskSetManager 的名称，可能会参与到公平调度算法中。
- parent：TaskSetManager 的父 Pool。
- totalResultSize：所有 Task 执行结果的总大小。
- calculatedTasks：计算过的 Task 数量。
- taskSetBlacklistHelperOpt：TaskSetManager 所管理的 TaskSet 的 Executor 或节点的黑名单。
- runningTasksSet：正在运行的 Task 的集合。
- runningTasks：正在运行的 Task 的数量。runningTasks 的值实际是通过调用 runningTasksSet 的 size 方法获取的。
- isZombie：当 TaskSetManager 所管理的 TaskSet 中的所有 Task 都执行成功了，不再有更多的 Task 尝试被启动时，就处于“僵尸”状态。例如，每个 Task 至少有一次尝试成功，或者 TaskSet 被舍弃了，TaskSetManager 将会进入“僵尸”状态。直到所有的 Task 都运行成功为止，TaskSetManager 将一直保持在“僵尸”状态。TaskSet-Manager 的“僵尸”状态并不是无用的，在这种状态下 TaskSetManager 将继续跟踪、记录正在运行的 Task。
- pendingTasksForExecutor：每个 Executor 上待处理的 Task 的集合，即 Executor 的身份标识与待处理 Task 的身份标识的集合之间的映射关系。
- pendingTasksForHost：每个 Host 上待处理的 Task 的集合，即 Host 与待处理 Task 的身份标识的集合之间的映射关系。
- pendingTasksForRack：每个机架上待处理的 Task 的集合，即机架与待处理 Task 的身份标识的集合之间的映射关系。
- pendingTasksWithNoPrefs：没有任何本地性偏好的待处理 Task 的身份标识的集合。
- allPendingTasks：所有待处理的 Task 的身份标识的集合。
- speculatableTasks：能够进行推断执行的 Task 的身份标识的集合。这类 Task 占所有 Task 的比例非常小。所谓推断执行，是指当 Task 的一次尝试运行非常缓慢，根

据推断，如果此时可以另起一次尝试运行，后来的尝试运行也比原先的尝试运行要快。

- taskInfos：Task 的身份标识与 Task 尝试的信息（如启动时间、完成时间等）之间的映射关系。
- EXCEPTION_PRINT_INTERVAL：异常打印到日志的时间间隔。可通过 spark.logging.exceptionPrintInterval 属性进行配置，默认为 10000。
- recentExceptions：类型为 HashMap[String, (Int, Long)]，用于缓存异常信息、异常次数及最后发生此异常的时间之间的映射关系。此属性与 EXCEPTION_PRINT_INTERVAL 配合使用。
- epoch：即 MapOutputTracker 的 epoch，用于故障转移。
- myLocalityLevels：Task 的本地性级别（7.6.4 节将详细介绍）的数组。MyLocalityLevels 是通过调用 computeValidLocalityLevels 方法（见代码清单 7-64）获取的。
- localityWaits：与 myLocalityLevels 中的每个本地性级别相对应，表示对应本地性级别的等待时间。localityWaits 实际是对 myLocalityLevels 中的每个本地性级别应用 getLocalityWait 方法（见代码清单 7-65）获取的。
- currentLocalityIndex：当前的本地性级别在 myLocalityLevels 中的索引。
- lastLaunchTime：在当前的本地性级别上运行 Task 的时间。
- emittedTaskSizeWarning：当发现序列化后的 Task 的大小超过了 100KB，此属性将被设置为 true，并且打印警告级别的日志。

了解了 TaskSetManager 的属性，现在来看看 TaskSetManager 是如何实现推断执行、本地性计算、资源分配的。

7.6.3 调度池与推断执行

在 Hadoop 2.x.x 版本中，当一个应用向 YARN 集群提交作业后，此作业的多个任务由于负载不均衡、资源分布不均等原因都会导致各个任务运行完成的时间不一致，甚至会出现一个 Task 尝试明显慢于同一作业的其他 Task 尝试的情况。如果对这种情况不加优化，最慢的 Task 尝试最终会拖慢整个作业的整体执行进度。好在 mapreduce 框架提供了任务推断执行机制，当有必要时就启动一个备份任务。最终会采用备份任务和原任务中率先执行完的结果作为最终结果。

与 Hadoop 类似，Spark 应用向 Spark 集群提交作业后，也会因为相似的原因导致出现慢任务拖慢整个作业执行进度的问题。为了解决这些问题，Pool 和 TaskSetManager 提供了 Spark 任务推断执行的实现。Pool 和 TaskSetManager 中对推断执行的操作分为两类：一类是可推断任务的检测与缓存；另一类是从缓存中找到可推断任务进行推断执行。Pool 的 checkSpeculatableTasks 方法（见代码清单 7-52）和 TaskSetManager 的 checkSpeculatableTasks 方法（见代码清单 7-62）实现了按照深度遍历算法对可推断任务的检测与缓存。TaskSet-

Manager 的 dequeueSpeculativeTask 方法（见代码清单 7-71）则实现了从缓存中找到可推断任务进行推断执行。

1. checkSpeculatableTasks

checkSpeculatableTasks 方法（见代码清单 7-62）用于检查当前 TaskSetManager 中是否有需要推断的任务。

代码清单7-62 checkSpeculatableTasks的实现

```
override def checkSpeculatableTasks(minTimeToSpeculation: Int): Boolean = {
  if (isZombie || numTasks == 1) {
    return false // 没有可以推断的Task
  }
  var foundTasks = false
  val minFinishedForSpeculation = (SPECULATION_QUANTILE * numTasks).floor.toInt
  logDebug("Checking for speculative tasks: minFinished = " + minFinished
      ForSpeculation)
  if (tasksSuccessful >= minFinishedForSpeculation && tasksSuccessful > 0) {
    val time = clock.getTimeMillis()
    val durations = taskInfos.values.filter(_.successful).map(_.duration).toArray
    Arrays.sort(durations)
    val medianDuration = durations(min((0.5 * tasksSuccessful).round.toInt,
        durations.length - 1))
    val threshold = max(SPECULATION_MULTIPLIER * medianDuration, minTime-
        ToSpeculation)
    logDebug("Task length threshold for speculation: " + threshold)
    for ((tid, info) <- taskInfos) { // 遍历taskInfos，寻找符合推断条件的Task
      val index = info.index
      if (!successful(index) && copiesRunning(index) == 1 && info.timeRunning(time)
          > threshold &&
        !speculatableTasks.contains(index)) {
        logInfo(
          "Marking task %d in stage %s (on %s) as speculatable because it ran more
          .      than %.0f ms"
            .format(index, taskSet.id, info.host, threshold))
        speculatableTasks += index
        foundTasks = true
      }
    }
  }
  foundTasks
}
```

根据代码清单 7-62，checkSpeculatableTasks 方法的执行步骤如下。

1）如果 TaskSetManager 处于“僵尸”状态且 TaskSet 只包含一个 Task，那么返回 false（即没有可以推断的 Task）。

2）计算进行推断的最小完成任务数量（minFinishedForSpeculation），即 SPECULATION_QUANTILE（开始推断的任务分数）与 numTasks（TaskSet 包含的 Task 的数量）的乘积向下取整。

3）如果执行成功的 Task 数量（tasksSuccessful）大于等于 minFinishedForSpeculation

并且 tasksSuccessful 大于 0，就进入下一步，否则返回 false。

4）找出 taskInfos 中执行成功的任务尝试信息（TaskInfo）中执行时间处于中间的时间 medianDuration。

5）计算进行推断的最小时间（threshold），即 SPECULATION_MULTIPLIER 与 medianDuration 的乘积和 minTimeToSpeculation 中的最大值。

6）遍历 taskInfos，将符合推断条件的 Task 在 TaskSet 中的索引放入 speculatableTasks 中。推断条件包括还未执行成功、复制运行数为一、运行时间大于 threshold。如果有 Task 的索引被放入了 speculatableTasks，那么返回 true。

2. dequeueSpeculativeTask

dequeueSpeculativeTask 方法（见代码清单 7-63）用于根据指定的 Host、Executor 和本地性级别，从可推断的 Task 中找出可推断的 Task 在 TaskSet 中的索引和相应的本地性级别。

代码清单7-63　找出可推断执行的Task

```
protected def dequeueSpeculativeTask(execId: String, host: String, locality:
    TaskLocality.Value)
  : Option[(Int, TaskLocality.Value)] =
{
  speculatableTasks.retain(index => !successful(index)) // 移除已经完成的Task
  def canRunOnHost(index: Int): Boolean = {
    !hasAttemptOnHost(index, host) &&
      !isTaskBlacklistedOnExecOrNode(index, execId, host)
  }

  if (!speculatableTasks.isEmpty) {
    for (index <- speculatableTasks if canRunOnHost(index)) {
      val prefs = tasks(index).preferredLocations
      val executors = prefs.flatMap(_ match {
        case e: ExecutorCacheTaskLocation => Some(e.executorId)
        case _ => None
      });
      if (executors.contains(execId)) { // 找到了在指定的Executor上推断执行的Task
        speculatableTasks -= index
        return Some((index, TaskLocality.PROCESS_LOCAL))
      }
    }

    if (TaskLocality.isAllowed(locality, TaskLocality.NODE_LOCAL)) {
      for (index <- speculatableTasks if canRunOnHost(index)) {
        val locations = tasks(index).preferredLocations.map(_.host)
        if (locations.contains(host)) { // 找到了在本地节点上推断执行的Task
          speculatableTasks -= index
          return Some((index, TaskLocality.NODE_LOCAL))
        }
      }
    }

    if (TaskLocality.isAllowed(locality, TaskLocality.NO_PREF)) {
```

```
        for (index <- speculatableTasks if canRunOnHost(index)) {
          val locations = tasks(index).preferredLocations
          if (locations.size == 0) {   // 对于没有本地性偏好的Task，让它在指定的Executor
              上推断执行
            speculatableTasks -= index
            return Some((index, TaskLocality.PROCESS_LOCAL))
          }
        }
      }

      if (TaskLocality.isAllowed(locality, TaskLocality.RACK_LOCAL)) {
        for (rack <- sched.getRackForHost(host)) {
          for (index <- speculatableTasks if canRunOnHost(index)) {
            val racks = tasks(index).preferredLocations.map(_.host).flatMap(sched.
                getRackForHost)
            if (racks.contains(rack)) { // 找到了本地机架上推断执行的Task
              speculatableTasks -= index
              return Some((index, TaskLocality.RACK_LOCAL))
            }
          }
        }
      }

      if (TaskLocality.isAllowed(locality, TaskLocality.ANY)) {
        for (index <- speculatableTasks if canRunOnHost(index)) {
          speculatableTasks -= index // 找到可以在任何节点、机架上推断执行的Task
          return Some((index, TaskLocality.ANY))
        }
      }
    }
    None
  }
```

根据代码清单 7-63，dequeueSpeculativeTask 方法的执行步骤如下。

1）从 speculatableTasks 中移除已经完成的 Task，保留还未完成的 Task。

2）对于 speculatableTasks 中的所有未在指定的 Host 上尝试运行，且指定的 Host 和 Executor 不在黑名单的所有 Task 进行以下处理：

① 获取 Task 偏好的 Executor；

② 如果 Task 偏好的 Executor 中包含指定的 Executor，那么将此 Task 的索引从 speculatableTasks 中移除，并返回此 Task 的索引与 PROCESS_LOCAL 的对偶。

3）如果指定的本地性级别小于等于 NODE_LOCAL，对 speculatableTasks 中所有未在指定的 Host 上尝试运行，且指定的 Host 和 Executor 不在黑名单的所有 Task 进行以下处理：

① 获取 Task 偏好的 Host；

② 如果 Task 偏好的 Host 中包含指定的 Host，那么将此 Task 的索引从 speculatableTasks 中移除，并返回此 Task 的索引与 NODE_LOCAL 的对偶。

4）如果指定的本地性级别小于等于 NO_PREF，对于 speculatableTasks 中的所有未

在指定的 Host 上尝试运行，且指定的 Host 和 Executor 不在黑名单的所有 Task 进行以下处理：

① 获取 Task 偏好的位置；

② 如果 Task 没有偏好的位置信息，那么将此 Task 的索引从 speculatableTasks 中移除，并返回此 Task 的索引与 PROCESS_LOCAL 的对偶。

5）如果指定的本地性级别小于等于 RACK_LOCAL，对于 speculatableTasks 中的所有未在指定的 Host 上尝试运行，且指定的 Host 和 Executor 不在黑名单的所有 Task 进行以下处理：

① 获取 Task 偏好的机架；

② 如果 Task 偏好的机架中包含指定的 Host 所在的机架，那么将此 Task 的索引从 speculatableTasks 中移除，并返回此 Task 的索引与 RACK_LOCAL 的对偶。

6）如果指定的本地性级别小于等于 ANY，对于 speculatableTasks 中的所有未在指定的 Host 上尝试运行，且指定的 Host 和 Executor 不在黑名单的所有 Task 进行以下处理：

① 将此 Task 的索引从 speculatableTasks 中移除；

② 返回此 Task 的索引与 ANY 的对偶。

7.6.4 Task 本地性

与 Hadoop 类似，Spark 对任务的处理也要考虑数据的本地性（Locality），好的数据本地性能够大幅减少节点间的数据传输，提升程序执行效率。Spark 目前支持五种本地性级别，由高到低分别为：PROCESS_LOCAL(本地进程)，NODE_LOCAL(本地节点)，NO_PREF(没有偏好)，RACK_LOCAL（本地机架），ANY（任何）。

Task 本地性的分配优先考虑有较高的本地性的级别，否则分配较低的本地性级别，直到 ANY。TaskSet 可以有一到多个本地性级别，但在给 Task 分配本地性时只能是其中的一个。TaskSet 中的所有 Task 都具有相同的允许使用的本地性级别，但在运行期可能因为资源不足、运行时间等因素，导致同一 TaskSet 中的各个 Task 的本地性级别可能不同。

TaskSet 中实现的本地性操作包括对 TaskSet 的本地性级别进行计算、获取某个本地性级别的等待时间、给 Task 分配资源时获取允许的本地性级别等。

1. computeValidLocalityLevels

computeValidLocalityLevels 方法（见代码清单 7-64）用于计算有效的本地性级别，这样就可以将 Task 按照本地性级别，由高到低地分配给允许的 Executor。

代码清单7-64 计算有效的本地性级别

```
private def computeValidLocalityLevels(): Array[TaskLocality.TaskLocality] = {
  import TaskLocality.{PROCESS_LOCAL, NODE_LOCAL, NO_PREF, RACK_LOCAL, ANY}
  val levels = new ArrayBuffer[TaskLocality.TaskLocality]
  if (!pendingTasksForExecutor.isEmpty && getLocalityWait(PROCESS_LOCAL) != 0 &&
      pendingTasksForExecutor.keySet.exists(sched.isExecutorAlive(_))) {
```

```
    levels += PROCESS_LOCAL // 允许的本地性级别里包括PROCESS_LOCAL
  }
  if (!pendingTasksForHost.isEmpty && getLocalityWait(NODE_LOCAL) != 0 &&
      pendingTasksForHost.keySet.exists(sched.hasExecutorsAliveOnHost(_))) {
    levels += NODE_LOCAL // 允许的本地性级别里包括NODE_LOCAL
  }
  if (!pendingTasksWithNoPrefs.isEmpty) {
    levels += NO_PREF // 允许的本地性级别里包括NO_PREF
  }
  if (!pendingTasksForRack.isEmpty && getLocalityWait(RACK_LOCAL) != 0 &&
      pendingTasksForRack.keySet.exists(sched.hasHostAliveOnRack(_))) {
    levels += RACK_LOCAL // 允许的本地性级别里包括RACK_LOCAL
  }
  levels += ANY // 允许的本地性级别里增加ANY
  logDebug("Valid locality levels for " + taskSet + ": " + levels.mkString(", "))
  levels.toArray // 返回所有允许的本地性级别
}
```

根据代码清单 7-64，computeValidLocalityLevels 方法的执行步骤如下。

1）如果存在 Executor 上待处理的 Task 的集合（即 pendingTasksForExecutor 不为空）且 PROCESS_LOCAL 级别的等待时间不为 0，还存在已被激活的 Executor（即 pendingTasksForExecutor 中的 ExecutorId 有存在于 TaskSchedulerImpl 的 executorIdToRunningTaskIds 中的），那么允许的本地性级别里包括 PROCESS_LOCAL。

2）如果存在 Host 上待处理的 Task 的集合（即 pendingTasksForHost 不为空）且 NODE_LOCAL 级别的等待时间不为 0，除此以外，Host 上存在已被激活的 Executor（即 pendingTasksForHost 中的 Host 有存在于 TaskSchedulerImpl 的 hostToExecutors 中的），那么允许的本地性级别里包括 NODE_LOCAL。

3）如果存在没有任何本地性偏好的待处理 Task，那么允许的本地性级别里包括 NO_PREF。

4）如果存在机架上待处理的 Task 的集合（即 pendingTasksForRack 不为空）且 RACK_LOCAL 级别的等待时间不为 0，除此以外，机架上存在已被激活的 Executor（即 pendingTasksForRack 中的机架有存在于 TaskSchedulerImpl 的 hostsByRack 中的），那么允许的本地性级别里包括 RACK_LOCAL。

5）允许的本地性级别里增加 ANY。

6）返回所有允许的本地性级别。

2. getLocalityWait

getLocalityWait 方法（见代码清单 7-65）用于获取某个本地性级别的等待时间。

代码清单7-65 获取某个本地性级别的等待时间

```
private def getLocalityWait(level: TaskLocality.TaskLocality): Long = {
  val defaultWait = conf.get("spark.locality.wait", "3s") // 获取默认的等待时间
  val localityWaitKey = level match { // 根据本地性级别匹配到对应的配置属性
    case TaskLocality.PROCESS_LOCAL => "spark.locality.wait.process"
```

```
      case TaskLocality.NODE_LOCAL => "spark.locality.wait.node"
      case TaskLocality.RACK_LOCAL => "spark.locality.wait.rack"
      case _ => null
    }
    if (localityWaitKey != null) { // 获取本地性级别对应的等待时间
      conf.getTimeAsMs(localityWaitKey, defaultWait)
    } else {
      0L
    }
  }
```

根据代码清单 7-65，getLocalityWait 方法的执行步骤如下。

1）获取默认的等待时间。默认的等待时间可以通过 spark.locality.wait 属性配置，默认为 3 秒。

2）根据本地性级别匹配到对应的配置属性。PROCESS_LOCAL 级别的配置属性为 spark.locality.wait.process, NODE_LOCAL 级别的配置属性为 spark.locality.wait.node, RACK_LOCAL 级别的配置属性为 spark.locality.wait.rack。其他级别没有配置属性。

3）使用第 2）步得到的属性名称，获取本地性级别对应的等待时间。如果第 2）步没有得到属性名称，那么将返回 0。

3. getLocalityIndex

getLocalityIndex 方法（见代码清单 7-66）用于从 myLocalityLevels 中找出指定的本地性级别所对应的索引。

代码清单7-66　获取本地性级别所对应的索引

```
def getLocalityIndex(locality: TaskLocality.TaskLocality): Int = {
  var index = 0
  while (locality > myLocalityLevels(index)) {
    index += 1
  }
  index
}
```

4. getAllowedLocalityLevel

getAllowedLocalityLevel 方法（见代码清单 7-67）用于获取允许的本地性级别。

代码清单7-67　获取允许的本地性级别

```
private def getAllowedLocalityLevel(curTime: Long): TaskLocality.TaskLocality = {
  while (currentLocalityIndex < myLocalityLevels.length - 1) {
    val moreTasks = myLocalityLevels(currentLocalityIndex) match { // 查找本地性级
        别是否有Task要运行
      case TaskLocality.PROCESS_LOCAL => moreTasksToRunIn(pendingTasksForExecutor)
      case TaskLocality.NODE_LOCAL => moreTasksToRunIn(pendingTasksForHost)
      case TaskLocality.NO_PREF => pendingTasksWithNoPrefs.nonEmpty
      case TaskLocality.RACK_LOCAL => moreTasksToRunIn(pendingTasksForRack)
    }
    if (!moreTasks) {
```

```
        lastLaunchTime = curTime // 没有Task需要处理，则将最后的运行时间设置为curTime
        logDebug(s"No tasks for locality level ${myLocalityLevels(currentLocalityI
            ndex)}, " +
          s"so moving to locality level ${myLocalityLevels(currentLocalityIndex + 1)}")
        currentLocalityIndex += 1
      } else if (curTime - lastLaunchTime >= localityWaits(currentLocalityIndex)) {
        lastLaunchTime += localityWaits(currentLocalityIndex) // 跳入更低的本地性级别
        logDebug(s"Moving to ${myLocalityLevels(currentLocalityIndex + 1)} after
            waiting for " +
          s"${localityWaits(currentLocalityIndex)}ms")
        currentLocalityIndex += 1
      } else {
        return myLocalityLevels(currentLocalityIndex) // 返回当前本地性级别
      }
    }
    myLocalityLevels(currentLocalityIndex) // 未能找到允许的本地性级别，那么返回最低的本地性
        级别
  }
```

根据代码清单 7-67，getAllowedLocalityLevel 方法的执行步骤如下。

1）按照索引由高到低从 myLocalityLevels 读取本地性级别，然后执行以下操作。

① 调用 moreTasksToRunIn 方法（见代码清单 7-76）判断本地性级别对应的待处理 Task 的缓存结构中是否有 Task 需要处理。

② 如果没有 Task 需要处理，则将最后的运行时间设置为 curTime。

③ 如果有 Task 需要处理且 curTime 与最后运行时间的差值大于当前本地性级别的等待时间，则将最后的运行时间增加当前本地性级别的等待时间（这样实际将直接跳入更低的本地性级别）。

④ 如果有 Task 需要处理且 curTime 与最后运行时间的差值小于等于当前本地性级别的等待时间，则返回当前本地性级别。

2）如果上一步未能找到允许的本地性级别，那么返回最低的本地性级别。

getAllowedLocalityLevel 方法的执行步骤也可以用图 7-11 来表示。

经过对 Spark 任务本地性的分析后，读者可能觉得这样的代码实现有些过于复杂，并且在获取本地性级别的时候竟然每次都要等待一段本地性级别的等待时长，这种实现未免太过奇怪。正如刚开始说的，任何任务都希望被分配到可以从本地读取数据的节点上，以得到最大的性能提升。然而每个任务的运行时长都不是事先可以预料的，当一个任务在分配时，如果没有满足最佳本地性（PROCESS_LOCAL）的资源，而一直固执地期盼得到最佳的资源，很有可能被已经占用最佳资源但是运行时间很长的任务耽搁，所以这些代码实现了当没有最佳本地性时，退而求其次，选择稍微差点的资源。

7.6.5 TaskSetManager 的常用方法

TaskSetManager 除提供了 Task 推断和本地性的方法外，还有很多方法。限于篇幅，本节将介绍 TaskSetManager 中其他最常用的方法。

仅供非商业用途或交流学习使用

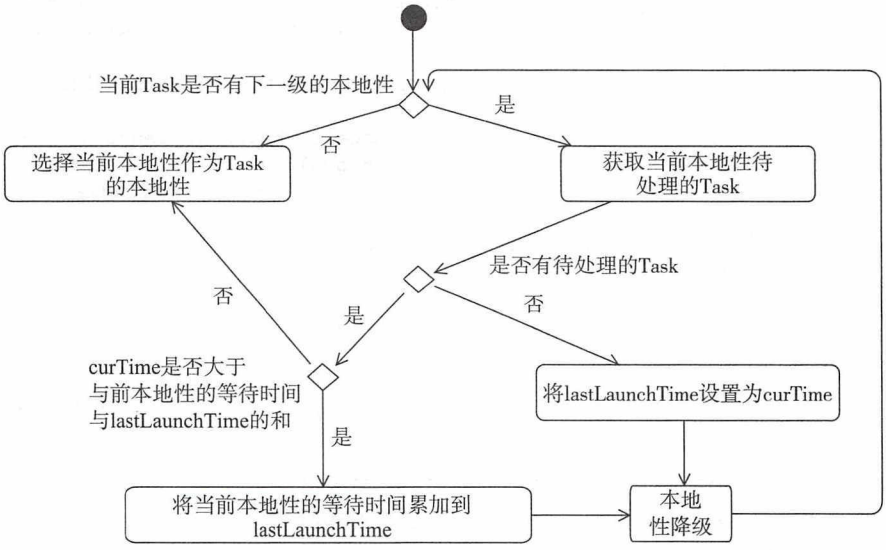


图 7-11　获取允许的本地性级别

1. addPendingTask

addPendingTask 方法（见代码清单 7-68）用于将待处理 Task 的索引按照 Task 的偏好位置，添加到 pendingTasksForExecutor、pendingTasksForHost、pendingTasksForRack、pendingTasksWithNoPrefs、allPendingTasks 等缓存中。



代码清单7-68　添加待处理任务

```
private def addPendingTask(index: Int) {
  for (loc <- tasks(index).preferredLocations) {
    loc match {
      case e: ExecutorCacheTaskLocation =>
        pendingTasksForExecutor.getOrElseUpdate(e.executorId, new ArrayBuffer) += index
      case e: HDFSCacheTaskLocation =>
        val exe = sched.getExecutorsAliveOnHost(loc.host)
        exe match {
          case Some(set) =>
            for (e <- set) {
              pendingTasksForExecutor.getOrElseUpdate(e, new ArrayBuffer) += index
            }
            logInfo(s"Pending task $index has a cached location at ${e.host} " +
              ", where there are executors " + set.mkString(","))
          case None => logDebug(s"Pending task $index has a cached location at ${e.
              host} " +
              ", but there are no executors alive there.")
        }
      case _ =>
    }
    pendingTasksForHost.getOrElseUpdate(loc.host, new ArrayBuffer) += index
    for (rack <- sched.getRackForHost(loc.host)) {
      pendingTasksForRack.getOrElseUpdate(rack, new ArrayBuffer) += index
```

仅供非商业用途或交流学习使用


```
    }
  }

  if (tasks(index).preferredLocations == Nil) {
    pendingTasksWithNoPrefs += index
  }

  allPendingTasks += index  // No point scanning this whole list to find the old task
      there
}
```

2. dequeueTaskFromList

dequeueTaskFromList 方法（见代码清单 7-69）用于从给定的 Task 列表中按照索引，从高到低找出满足条件（不在黑名单中、Task 的复制运行数等于 0、Task 没有成功）的 Task 的索引。

代码清单7-69 dequeueTaskFromList的实现

```
private def dequeueTaskFromList(
    execId: String,
    host: String,
    list: ArrayBuffer[Int]): Option[Int] = {
  var indexOffset = list.size
  while (indexOffset > 0) {
    indexOffset -= 1
    val index = list(indexOffset)
    if (!isTaskBlacklistedOnExecOrNode(index, execId, host)) {
      list.remove(indexOffset)
      if (copiesRunning(index) == 0 && !successful(index)) {
        return Some(index)
      }
    }
  }
  None
}
```

3. dequeueTask

dequeueTask 方法（见代码清单 7-70）用于根据指定的 Host、Executor 和本地性级别，找出要执行的 Task 的索引、相应的本地性级别及是否进行推断执行。

代码清单7-70 dequeueTask的实现

```
private def dequeueTask(execId: String, host: String, maxLocality: TaskLocality.Value)
  : Option[(Int, TaskLocality.Value, Boolean)] =
{
  for (index <- dequeueTaskFromList(execId, host, getPendingTasksForExecutor(exe
      cId))) {
    return Some((index, TaskLocality.PROCESS_LOCAL, false)) // 选择指定Executor的待
        处理Task
  }

  if (TaskLocality.isAllowed(maxLocality, TaskLocality.NODE_LOCAL)) {
```

仅供非商业用途或交流学习使用


```
      for (index <- dequeueTaskFromList(execId, host, getPendingTasksForHost(host))) {
        return Some((index, TaskLocality.NODE_LOCAL, false)) // 选择指定Host上的待处理Task
      }
    }
    if (TaskLocality.isAllowed(maxLocality, TaskLocality.NO_PREF)) {
      for (index <- dequeueTaskFromList(execId, host, pendingTasksWithNoPrefs)) {
        return Some((index, TaskLocality.PROCESS_LOCAL, false)) // 选择没有本地性偏好
            的待处理Task
      }
    }
    if (TaskLocality.isAllowed(maxLocality, TaskLocality.RACK_LOCAL)) {
      for {
        rack <- sched.getRackForHost(host)
        index <- dequeueTaskFromList(execId, host, getPendingTasksForRack(rack))
      } {
        return Some((index, TaskLocality.RACK_LOCAL, false)) // 选择指定Host所在机架
            上的待处理Task
      }
    }
    if (TaskLocality.isAllowed(maxLocality, TaskLocality.ANY)) {
      for (index <- dequeueTaskFromList(execId, host, allPendingTasks)) {
        return Some((index, TaskLocality.ANY, false)) // 选择在任意的本地性上的待处理
            Task
      }
    }

    dequeueSpeculativeTask(execId, host, maxLocality).map { // 选择可推断的Task及其本地性
      case (taskIndex, allowedLocality) => (taskIndex, allowedLocality, true)}
  }
```



根据代码清单 7-70，dequeueTask 方法的执行步骤如下。

1）从 pendingTasksForExecutor 中找出指定 Executor 的待处理 Task 中满足条件（不在黑名单中、Task 的复制运行数等于 0、Task 没有成功）的 Task 的索引，并返回此索引、PROCESS_LOCAL、非推断执行标记（false）的三元组。

2）如果指定的本地性级别小于等于 NODE_LOCAL，调用 dequeueTaskFromList 方法（见代码清单 7-69），从 pendingTasksForHost 中找出指定 Host 的待处理 Task 中满足条件（不在黑名单中、Task 的复制运行数等于 0、Task 没有成功）的 Task 的索引，并返回此索引、NODE_LOCAL、非推断执行标记（false）的三元组。

3）如果指定的本地性级别小于等于 NO_PREF，调用 dequeueTaskFromList 方法（见代码清单 7-69），从 pendingTasksWithNoPrefs 中找出满足条件（不在黑名单中、Task 的复制运行数等于 0、Task 没有成功）的 Task 的索引，并返回此索引、PROCESS _LOCAL、非推断执行标记（false）的三元组。

4）如果指定的本地性级别小于等于 RACK_LOCAL，调用 dequeueTaskFromList 方法（见代码清单 7-69），从 pendingTasksForRack 中找出指定 Host 所在机架的待处理 Task 中满足条件（不在黑名单中、Task 的复制运行数等于 0、Task 没有成功）的 Task 的索引，并返回此索引、RACK_LOCAL、非推断执行标记（false）的三元组。

仅供非商业用途或交流学习使用

5）如果指定的本地性级别小于等于ANY，调用dequeueTaskFromList方法（见代码清单7-69），从allPendingTasks中找出满足条件（不在黑名单中、Task的复制运行数等于0、Task没有成功）的Task的索引，并返回此索引、ANY、非推断执行标记（false）的三元组。

6）调用dequeueSpeculativeTask方法（见代码清单7-63），从可推断的Task中找出可推断的Task的索引和相应的本地性级别，并返回可推断的Task的索引、相应的本地性级别、推断执行标记（true）的三元组。

4. addRunningTask与removeRunningTask

addRunningTask方法（见代码清单7-71）用于向runningTasksSet中添加Task的身份标识，并调用TaskSetManager的父调度池的increaseRunningTasks方法（见代码清单7-54），增加父调度池及祖父调度池中记录的当前正在运行的任务数量。而removeRunningTask方法则相反（见代码清单7-71）。

代码清单7-71 增加或减少正在运行的任务数量

```
def addRunningTask(tid: Long) {
  if (runningTasksSet.add(tid) && parent != null) {
    parent.increaseRunningTasks(1)
  }
}
def removeRunningTask(tid: Long) {
  if (runningTasksSet.remove(tid) && parent != null) {
    parent.decreaseRunningTasks(1)
  }
}
```



5. maybeFinishTaskSet

maybeFinishTaskSet方法（见代码清单7-72）用于当TaskSet可能已经完成的时候进行一些清理工作。TaskSchedulerImpl的taskSetFinished方法的实现很简单，留给感兴趣的读者自行研究。

代码清单7-72 maybeFinishTaskSet的实现

```
private def maybeFinishTaskSet() {
  if (isZombie && runningTasks == 0) {
    sched.taskSetFinished(this)
  }
}
```

6. resourceOffer

resourceOffer方法（见代码清单7-73）用于给Task按照本地性分配资源。

代码清单7-73 给Task分配资源

```
def resourceOffer(
    execId: String,
    host: String,
    maxLocality: TaskLocality.TaskLocality)
```

仅供非商业用途或交流学习使用


```
    : Option[TaskDescription] =
  {
    val offerBlacklisted = taskSetBlacklistHelperOpt.exists { blacklist =>
      blacklist.isNodeBlacklistedForTaskSet(host) ||
        blacklist.isExecutorBlacklistedForTaskSet(execId)
    }
    if (!isZombie && !offerBlacklisted) {
      val curTime = clock.getTimeMillis()
      var allowedLocality = maxLocality
      if (maxLocality != TaskLocality.NO_PREF) { // 计算允许的本地性级别
        allowedLocality = getAllowedLocalityLevel(curTime)
        if (allowedLocality > maxLocality) {
          allowedLocality = maxLocality
        }
      }
      // 根据指定的Host、Executor和本地性级别，找到合适的Task
      dequeueTask(execId, host, allowedLocality).map { case ((index, taskLocality,
          speculative)) =>
        val task = tasks(index) // 根据要执行的Task的索引找到要执行的Task
        val taskId = sched.newTaskId() // 为Task生成新的身份标识
        copiesRunning(index) += 1 // 增加复制运行数
        val attemptNum = taskAttempts(index).size // 获取任务尝试号attemptNum
        val info = new TaskInfo(taskId, index, attemptNum, curTime,
          execId, host, taskLocality, speculative) // 创建Task尝试信息
        taskInfos(taskId) = info // 将Task的身份标识与TaskInfo的对应关系放入taskInfos
        // 将TaskInfo添加到taskAttempts中
        taskAttempts(index) = info :: taskAttempts(index)
        if (maxLocality != TaskLocality.NO_PREF) { // 获取Task的本地性偏好级别
          currentLocalityIndex = getLocalityIndex(taskLocality)
          lastLaunchTime = curTime
        }
        val startTime = clock.getTimeMillis()
        val serializedTask: ByteBuffer = try { // 序列化Task
          Task.serializeWithDependencies(task, sched.sc.addedFiles, sched.sc.addedJars,
              ser)
        } catch {
          case NonFatal(e) =>
            val msg = s"Failed to serialize task $taskId, not attempting to retry it."
            logError(msg, e)
            abort(s"$msg Exception during serialization: $e")
            throw new TaskNotSerializableException(e)
        }
        if (serializedTask.limit > TaskSetManager.TASK_SIZE_TO_WARN_KB * 1024 &&
          !emittedTaskSizeWarning) {
          emittedTaskSizeWarning = true
          logWarning(s"Stage ${task.stageId} contains a task of very large size " +
              s"(${serializedTask.limit / 1024} KB). The maximum recommended task
                  size is " +
            s"${TaskSetManager.TASK_SIZE_TO_WARN_KB} KB.")
        }
        addRunningTask(taskId) // 将Task放入正在运行的Task中

        val taskName = s"task ${info.id} in stage ${taskSet.id}" // 生成Task的名称
        logInfo(s"Starting $taskName (TID $taskId, $host, executor ${info.executorId}, " +
          s"partition ${task.partitionId}, $taskLocality, ${serializedTask.limit}
              bytes)")
```



仅供非商业用途或交流学习使用


```
        sched.dagScheduler.taskStarted(task, info)
        new TaskDescription(taskId = taskId, attemptNumber = attemptNum, execId,
          taskName, index, serializedTask) // 创建并返回TaskDescription对象
      }
    } else {
      None
    }
  }
```

根据代码清单7-73，resourceOffer方法在TaskSetManager处于“僵尸”状态并且分配Task的Host和Executor在黑名单中时直接返回None，否则执行如下步骤。

1）计算允许的本地性级别。如果最大本地性级别（即maxLocality）为NO_PREF，则允许的本地性级别为NO_PREF。如果maxLocality不是NO_PREF，则允许的本地性级别为maxLocality和调用getAllowedLocalityLevel获取的本地性级别中较小的本地性级别。

2）调用dequeueTask方法（见代码清单7-70），根据指定的Host、Executor和本地性级别，找出三元组（包括要执行的Task的索引、相应的本地性级别、是否推断执行的标记），并对三元组进行如下操作。

① 根据要执行的Task的索引找到要执行的Task。

② 为Task生成新的身份标识。

③ 将Task对应的copiesRunning信息加一，即增加复制运行数。

④ 获取任务尝试号attemptNum。

⑤ 创建任务尝试信息（TaskInfo）。

⑥ 将Task的身份标识与TaskInfo的对应关系放入taskInfos。

⑦ 将TaskInfo添加到taskAttempts中Task对应的TaskInfo列表中。

⑧ 如果maxLocality不是NO_PREF，那么调用getLocalityIndex方法（见代码清单7-66），获取任务的本地性偏好级别在myLocalityLevels中的索引，并将最后一次运行时间设置为当前系统时间。

⑨ 将Task、用户添加的Jar包及其他文件序列化，得到需要经过网络传输的序列化Task。

⑩ 如果序列化Task的大小超过了100KB，且emittedTaskSizeWarning仍然为false，则将emittedTaskSizeWarning设置为true并且打印警告日志。

⑪ 调用addRunningTask方法（见代码清单7-71），向runningTasksSet中添加Task的身份标识，并增加父调度池及祖父调度池中记录的当前正在运行的任务数量。

⑫ 生成Task的名称，格式为："task $index.$attemptNumber in stage $stageId. $stage AttemptId"。

⑬ 调用DAGScheduler的taskStarted方法向DAGSchedulerEventProcessLoop投递BeginEvent事件。

⑭ 创建并返回TaskDescription对象。



仅供非商业用途或交流学习使用

7. executorLost

executorLost 方法在发生 Executor 丢失的情况下被调用，其实现如代码清单 7-74 所示。

代码清单7-74 executorLost的实现

```
override def executorLost(execId: String, host: String, reason: Executor LossReason) {
  if (tasks(0).isInstanceOf[ShuffleMapTask] && !env.blockManager.externalShuffle
      ServiceEnabled) {
    for ((tid, info) <- taskInfos if info.executorId == execId) {
      val index = taskInfos(tid).index
      if (successful(index)) {
        successful(index) = false // 将此Task标记为未成功
        copiesRunning(index) -= 1 // 将此Task的复制运行数量减一
        tasksSuccessful -= 1      // 将当前TaskSetManager中成功执行的Task数量减一
        addPendingTask(index)     // 将此Task添加到待处理的Task中
        sched.dagScheduler.taskEnded(
          tasks(index), Resubmitted, null, Seq.empty, info) // 告知DAGScheduler重
              新提交Task
      }
    }
  }
  for ((tid, info) <- taskInfos if info.running && info.executorId == execId) {
    val exitCausedByApp: Boolean = reason match { // 获取Executor丢失的具体原因是否
        是由应用程序引起的
      case exited: ExecutorExited => exited.exitCausedByApp
      case ExecutorKilled => false
      case _ => true
    }
    handleFailedTask(tid, TaskState.FAILED, ExecutorLostFailure(info.executorId,
        exitCausedByApp,
      Some(reason.toString))) // 对失败的Task进行处理
  }
  recomputeLocality() // 重新计算本地性级别
}
```

根据代码清单 7-74，executorLost 方法的执行步骤如下。

1）如果 TaskSetManager 管理的 TaskSet 中的 Task 为 ShuffleMapTask，并且应用没有提供外部的 Shuffle 服务，那么对 taskInfos 中的所有在指定 Executor 上执行成功的 Task 执行以下操作。

① 将此 Task 标记为未成功。

② 将此 Task 的复制运行数量减一。

③ 将当前 TaskSetManager 中成功执行的 Task 数量减一。

④ 调用 addPendingTask 方法（见代码清单 7-68），将此 Task 的索引添加到 pendingTasksForExecutor、pendingTasksForHost、pendingTasksForRack、pendingTasksWithNoPrefs、allPendingTasks 等缓存中。

⑤ 调用 DAGScheduler 的 taskEnded 方法向 DAGSchedulerEventProcessLoop 发送 CompletionEvent。此时的 CompletionEvent 将携带 Resubmitted，避免 CompletionEvent 的关心者在 Stage 的所有 Task 都完成后认为当前的 Stage 也执行完成，而是让它们知道此 Task 将被重

仅供非商业用途或交流学习使用

新提交。

2）对 taskInfos 中的所有在身份标识为 execId 的 Executor 上正在运行的 Task 执行以下操作。

① 从 ExecutorLossReason 中获取 Executor 丢失的具体原因是否是由应用程序引起的。

② 调用 handleFailedTask 方法（由于此方法不属于主线逻辑，所以留给感兴趣的读者自行阅读）对失败的 Task 进行处理。

3）重新计算本地性级别。

8. tasksNeedToBeScheduledFrom

tasksNeedToBeScheduledFrom 方法（见代码清单 7-75）用于判断给定的待处理 Task 数组中是否有需要调度的 Task。

代码清单7-75 tasksNeedToBeScheduledFrom的实现

```
def tasksNeedToBeScheduledFrom(pendingTaskIds: ArrayBuffer[Int]): Boolean = {
  var indexOffset = pendingTaskIds.size
  while (indexOffset > 0) {
    indexOffset -= 1
    val index = pendingTaskIds(indexOffset)
    if (copiesRunning(index) == 0 && !successful(index)) {
      return true
    } else {
      pendingTaskIds.remove(indexOffset)
    }
  }
  false
}
```



根据代码清单 7-75，判断一个 Task 是否需要调度的条件是 Task 没有被复制运行且 Task 还未执行成功。

9. moreTasksToRunIn

moreTasksToRunIn 方法（见代码清单 7-76）用于判断待处理的 Task 集合（数据结构为 HashMap[String, ArrayBuffer[Int]]）中是否有需要调度的 Task。如果有，则返回 true，否则将此 key 与 Task 集合的映射关系从待处理的 Task 集合中移除并返回 false。

代码清单7-76 moreTasksToRunIn的实现

```
def moreTasksToRunIn(pendingTasks: HashMap[String, ArrayBuffer[Int]]): Boolean = {
  val emptyKeys = new ArrayBuffer[String]
  val hasTasks = pendingTasks.exists {
    case (id: String, tasks: ArrayBuffer[Int]) =>
      if (tasksNeedToBeScheduledFrom(tasks)) {
        true
      } else {
        emptyKeys += id
        false
      }
```

```
    }
    emptyKeys.foreach(id => pendingTasks.remove(id))
    hasTasks
  }
```

10. handleSuccessfulTask

handleSuccessfulTask 方法（见代码清单 7-77）用于对 Task 的执行结果（对于 map 任务而言，实际是任务状态）进行处理。

代码清单7-77 对Task的执行结果进行处理

```
def handleSuccessfulTask(tid: Long, result: DirectTaskResult[_]): Unit = {
  val info = taskInfos(tid)
  val index = info.index
  info.markFinished(TaskState.FINISHED) // 将taskInfos中缓存的TaskInfo标记为已经完成
  removeRunningTask(tid) // 将Task从正在运行的Task集合中移除
  // 将Task的执行结果交给DagScheduler处理
  sched.dagScheduler.taskEnded(tasks(index), Success, result.value(), result.
      accumUpdates, info)
  for (attemptInfo <- taskAttempts(index) if attemptInfo.running) { //"杀死"此Task
       正在运行的Task尝试
    logInfo(s"Killing attempt ${attemptInfo.attemptNumber} for task ${attemptInfo.
        id} " +
      s"in stage ${taskSet.id} (TID ${attemptInfo.taskId}) on ${attemptInfo.host} " +
      s"as the attempt ${info.attemptNumber} succeeded on ${info.host}")
    sched.backend.killTask(attemptInfo.taskId, attemptInfo.executorId, true)
  }
  if (!successful(index)) {
    tasksSuccessful += 1 // 增加运行成功的Task数量
    logInfo(s"Finished task ${info.id} in stage ${taskSet.id} (TID ${info.taskId})
        in" +
      s" ${info.duration} ms on ${info.host} (executor ${info.executorId})" +
      s" ($tasksSuccessful/$numTasks)")
    successful(index) = true // 将此任务的状态设置为true
    if (tasksSuccessful == numTasks) {
      isZombie = true // TaskSet中的所有Task都调度运行成功，于是将isZombie设置为true
    }
  } else {
    logInfo("Ignoring task-finished event for " + info.id + " in stage " + taskSet.
        id +
      " because task " + index + " has already completed successfully")
  }
  maybeFinishTaskSet() // 在TaskSet可能已经完成的时候进行一些清理工作
}
```

根据代码清单 7-77，handleSuccessfulTask 方法的执行步骤如下。

1）将 taskInfos 中缓存的 TaskInfo 标记为已经完成。

2）将 Task 的 ID 从正在运行的 Task 集合中移除，并且减少正在运行的 Task 的数量。

3）调用 DAGScheduler 的 taskEnded 方法（见代码清单 7-45），将 Task 的执行结果交给 DagScheduler 处理。

4）“杀死”此 Task 正在运行的 Task 尝试。

5）增加运行成功的 Task 数量，并且将 successful 中代表此任务的状态设置为 true。

6）如果成功运行的 Task 数量与 TaskSet 中的 Task 数量相同，说明此 TaskSet 中的所有 Task 都调度运行成功，于是将 isZombie 设置为 true。

7）调用 maybeFinishTaskSet 方法（见代码清单 7-72），在 TaskSet 可能已经完成的时候进行一些清理工作。

7.7 运行器后端接口 LauncherBackend

要介绍 LauncherBackend，首先应该介绍下 LauncherServer。当 Spark 应用程序没有在用户应用程序中运行，而是运行在单独的进程中时，用户可以在用户应用程序中使用 LauncherServer 与 Spark 应用程序通信。LauncherServer 将提供 Socket 连接的服务端，与 Spark 应用程序中的 Socket 连接的客户端通信。LauncherServer 的工作原理可以用图 7-12 来表示。

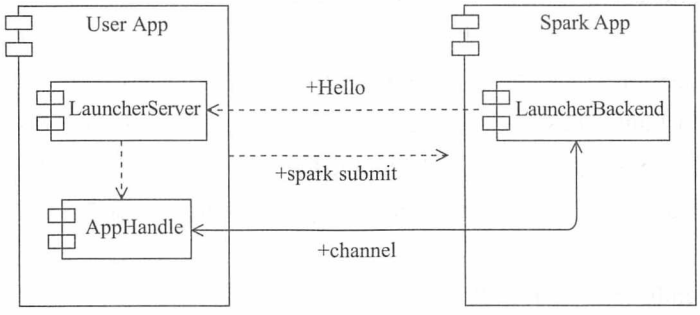


图 7-12 LauncherServer 的工作原理

由于 TaskSchedulerImpl 的底层依赖于 LauncherBackend，所以本节需要对 Launcher Backend 进行介绍。LauncherBackend 依赖于 BackendConnection，跟 Launcher Server 进行通信，LauncherBackend 的主要工作流程如图 7-13 所示。

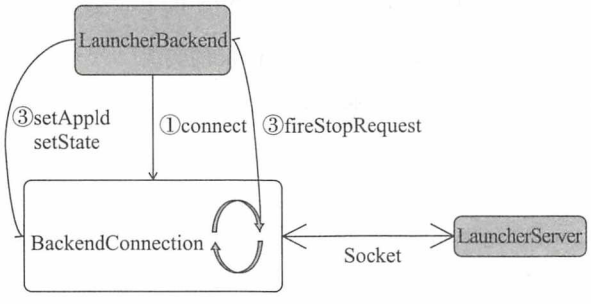


图 7-13 LauncherBackend 的主要工作流程

这里对图 7-13 中的各个记号进行说明。

记号①：调用 LauncherBackend 的 connect 方法创建 BackendConnection，并且创建线程执行 BackendConnection。构造 BackendConnection 的过程中，BackendConnection 会和 LauncherServer 之间建立起 Socket 连接。BackendConnection（实现了 java.lang.Runnable 接口）将不断从 Socket 连接中读取 LauncherServer 发送的数据。

记号②：调用 LauncherBackend 的 setAppId 方法（见代码清单 7-84）或 SetState 方法（见代码清单 7-85），通过 Socket 连接向 LauncherServer 发送 SetAppId 消息或 SetState 消息。

记号③：BackendConnection 从 Socket 连接中读取到 LauncherServer 发送的 Stop 消息，然后调用 LauncherBackend 的 fireStopRequest 方法停止请求。

下面将详细介绍 LauncherBackend 及其内部组件 BackendConnection 的实现。

7.7.1 BackendConnection 的实现

BackendConnection 是 LauncherBackend 的内部组件，用于保持与 LauncherServer 的 Socket 连接，并通过此 Socket 连接收发消息。BackendConnection 继承了 LauncherConnection，LauncherConnection 提供了维护连接和收发消息的基本实现。

BackendConnection 继承了抽象类 LauncherConnection 的属性，分别如下。

- socket：与 LauncherServer 的 Socket 服务端建立连接的 Socket 客户端。
- out：建立在 Socket 的输出流上的 ObjectOutputStream，用于向服务端发送消息。
- closed：Socket 客户端与 LauncherServer 的 Socket 服务端建立的连接是否已经关闭的状态。

有了对 BackendConnection 属性的了解，现在来看看 BackendConnection 提供的方法。

1. handle

handle 是 LauncherConnection 提供的用于处理 LauncherServer 发送的消息的抽象方法，其定义如下。

```
protected abstract void handle(Message msg) throws IOException;
```

BackendConnection 实现了 LauncherConnection 的 handle 方法，如代码清单 7-78 所示。

代码清单7-78　handle的实现

```
override protected def handle(m: Message): Unit = m match {
  case _: Stop =>
    fireStopRequest()

  case _ =>
    throw new IllegalArgumentException(s"Unexpected message type: ${m.getClass().
getName()}")
}
```

根据代码清单 7-78，BackendConnection 实现的 handle 方法只处理 Stop 这一种消息。

对于Stop消息，BackendConnection将调用外部类LauncherBackend的fireStopRequest方法（见代码清单7-86）停止Executor。

2. run

由于LauncherConnection实现了java.lang.Runnable接口，因此需要实现run方法。LauncherConnection的run方法（见代码清单7-79）用于从Socket客户端的输入流中读取LauncherServer发送的消息，并调用handle方法对消息进行处理。LauncherConnection的run方法同时也是一个模板方法。

代码清单7-79 LauncherConnection的run方法

```
@Override
public void run() {
  try {
    ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
    while (!closed) {
      Message msg = (Message) in.readObject();
      handle(msg);
    }
  } catch (EOFException eof) {
    try {
      close();
    } catch (Exception unused) {
      // no-op.
    }
  } catch (Exception e) {
    if (!closed) {
      LOG.log(Level.WARNING, "Error in inbound message handling.", e);
      try {
        close();
      } catch (Exception unused) {
        // no-op.
      }
    }
  }
}
```

3. send

此方法通过Socket客户端与LauncherServer的Socket服务端建立的连接向LauncherServer发送消息，其实现如代码清单7-80所示。

代码清单7-80 向LauncherServer发送消息

```
protected synchronized void send(Message msg) throws IOException {
  try {
    CommandBuilderUtils.checkState(!closed, "Disconnected.");
    out.writeObject(msg);
    out.flush();
  } catch (IOException ioe) {
    if (!closed) {
      LOG.log(Level.WARNING, "Error when sending message.", ioe);
```

```
        try {
          close();
        } catch (Exception unused) {
          // no-op.
        }
      }
      throw ioe;
    }
  }
```

4. close

此方法用于关闭 Socket 客户端与 LauncherServer 的 Socket 服务端建立的连接，其实现如代码清单 7-81 所示。

代码清单7-81 关闭连接

```
@Override
public void close() throws IOException {
  if (!closed) {
    synchronized (this) {
      if (!closed) {
        closed = true;
        socket.close();
      }
    }
  }
}
```

BackendConnection 重写了 LauncherConnection 的 close 方法，其实现如代码清单 7-82 所示。

代码清单7-82 BackendConnection的close方法

```
override def close(): Unit = {
  try {
    super.close()
  } finally {
    onDisconnected()
    _isConnected = false
  }
}
```

根据代码清单 7-82，BackendConnection 重写的 close 方法首先调用了父类 LauncherConnection 的 close 方法关闭 Socket 连接，然后调用外部类 LauncherBackend 的 onDisconnected 方法。由于 LauncherBackend 的 onDisconnected 是个空方法，因此 onDisconnected 并不会有额外的效果。

7.7.2 LauncherBackend 的实现

LauncherBackend 是 SchedulerBackend 与 LauncherServer 通信的组件，我们先来介绍

其属性信息，然后分析 LauncherBackend 的具体实现。

LauncherBackend 包含的属性如下。

- clientThread：读取与 LauncherServer 建立的 Socket 连接上的消息的线程。
- connection：即 BackendConnection 实例。
- lastState：LauncherBackend 的最后一次状态。lastState 的类型是枚举类型 SparkAppHandle.State。SparkAppHandle.State 共有未知（UNKNOWN）、已连接（CONNECTED）、已提交（SUBMITTED）、运行中（RUNNING）、已完成（FINISHED）、已失败（FAILED）、已被杀（KILLED）、丢失（LOST）等状态。
- _isConnected：clientThread 是否与 LauncherServer 已经建立了 Socket 连接的状态。

LauncherBackend 提供的方法如下。

1. connect

connect 方法（见代码清单 7-83）用于和 LauncherServer 建立连接。

代码清单7-83 建立连接

```
def connect(): Unit = {
  val port = sys.env.get(LauncherProtocol.ENV_LAUNCHER_PORT).map(_.toInt)
  val secret = sys.env.get(LauncherProtocol.ENV_LAUNCHER_SECRET)
  if (port != None && secret != None) { // 创建与LauncherServer的Socket服务端建立连
      接的Socket
    val s = new Socket(InetAddress.getLoopbackAddress(), port.get)
    connection = new BackendConnection(s)
    connection.send(new Hello(secret.get, SPARK_VERSION)) // 向LauncherServer发送
        Hello消息
    clientThread = LauncherBackend.threadFactory.newThread(connection)
    clientThread.start() // 创建并启动一个执行BackendConnection的run方法的线程
    _isConnected = true // 设置已连接的状态
  }
}
```

根据代码清单 7-83，connect 方法的执行步骤如下。

1）创建与 LauncherServer 的 Socket 服务端建立连接的 Socket。LauncherServer 的 Socket 端口和密钥可通过系统环境变量 _SPARK_LAUNCHER_PORT 和 _SPARK_LAUNCHER_SECRET 指定。

2）通过此连接向 LauncherServer 发送 Hello 消息。Hello 消息携带着应用程序的 Spark 版本号和密钥信息。

3）创建并启动一个执行 BackendConnection 的 run 方法的线程。

4）将 _isConnected 设置为 true。

2. setAppId

setAppId 方法（见代码清单 7-84）用于向 LauncherServer 发送 SetAppId 消息。SetAppId 消息携带着应用程序的身份标识。

代码清单7-84 向LauncherServer发送SetAppId消息

```
def setAppId(appId: String): Unit = {
  if (connection != null) {
    connection.send(new SetAppId(appId))
  }
}
```

3. setState

SetState 方法（见代码清单 7-85）用于向 LauncherServer 发送 SetState 消息。SetState 消息携带着 LauncherBackend 的最后一次状态。

代码清单7-85 向LauncherServer发送SetState消息

```
def setState(state: SparkAppHandle.State): Unit = {
  if (connection != null && lastState != state) {
    connection.send(new SetState(state))
    lastState = state
  }
}
```

4. isConnected

isConnected 方法返回 clientThread 是否与 LauncherServer 已经建立了 Socket 连接的状态。

```
def isConnected(): Boolean = _isConnected
```

5. onStopRequest

OnStopRequest 是 LauncherBackend 定义的处理 LauncherServer 的停止消息的抽象方法，代码定义如下。

```
protected def onStopRequest(): Unit
```

6. onDisconnected

onDisconnected 方法用于在关闭 Socket 客户端与 LauncherServer 的 Socket 服务端建立的连接时，进行一些额外的处理，但目前只是一个空方法。

```
protected def onDisconnected() : Unit = { }
```

7. fireStopRequest

fireStopRequest（见代码清单 7-86）用于启动一个调用 onStopRequest 方法的线程。

代码清单7-86 fireStopRequest的实现

```
private def fireStopRequest(): Unit = {
  val thread = LauncherBackend.threadFactory.newThread(new Runnable() {
    override def run(): Unit = Utils.tryLogNonFatalError {
      onStopRequest()
    }
  })
  thread.start()
}
```

7.8 调度后端接口 SchedulerBackend

SchedulerBackend 是 TaskScheduler 的调度后端接口。TaskScheduler 给 Task 分配资源实际是通过 SchedulerBackend 来完成的，SchedulerBackend 给 Task 分配完资源后将与分配给 Task 的 Executor 通信，并要求后者运行 Task。

7.8.1 SchedulerBackend 的定义

特质 SchedulerBackend 定义了所有调度后端接口的行为规范，其定义如代码清单 7-87 所示。

代码清单7-87 SchedulerBackend的定义

```
private[spark] trait SchedulerBackend {
  private val appId = "spark-application-" + System.currentTimeMillis
  def start(): Unit
  def stop(): Unit
  def reviveOffers(): Unit
  def defaultParallelism(): Int

  def killTask(taskId: Long, executorId: String, interruptThread: Boolean): Unit =
    throw new UnsupportedOperationException
  def isReady(): Boolean = true
  def applicationId(): String = appId
  def applicationAttemptId(): Option[String] = None
  def getDriverLogUrls: Option[Map[String, String]] = None
}
```

根据代码清单 7-87，SchedulerBackend 定义了以下成员。

- appId：与当前 Job 相关联的应用程序的身份标识。
- start：启动 SchedulerBackend，需要子类实现。
- stop：停止 SchedulerBackend，需要子类实现。
- reviveOffers：给调度池中的所有 Task 分配资源。
- defaultParallelism：获取 Job 的默认并行度。
- killTask：“杀死”指定的任务。可以通过设置 interruptThread 为 true 来中断任务执行线程。
- isReady：SchedulerBackend 是否准备就绪。
- applicationId：获取 appId。
- applicationAttemptId：当应用在 cluster 模式运行且集群管理器支持应用进行多次执行尝试时，此方法可以获取应用程序尝试的标识。当应用程序在 client 模式运行时，将不支持多次尝试，因此此方法不会获取到应用程序尝试的标识。



注意 SparkContext 的 _applicationId 属性和 _applicationAttemptId 属性是通过分别调用 TaskSchedulerImpl 的 applicationId 方法和 applicationAttemptId 方法获得的，而 TaskSchedulerImpl 的 applicationId 方法和 applicationAttemptId 方法实际又分别调用了 SchedulerBackend 的 applicationId 方法和 applicationAttemptId 方法。

- getDriverLogUrls：获取 Driver 日志的 Url。这些 Url 将被用于在 Spark UI 的 Executors 标签页中展示。

了解了 SchedulerBackend 的接口定义，现在一起来看看在 Spark 中有哪些 SchedulerBackend 的实现类，如图 7-14 所示。

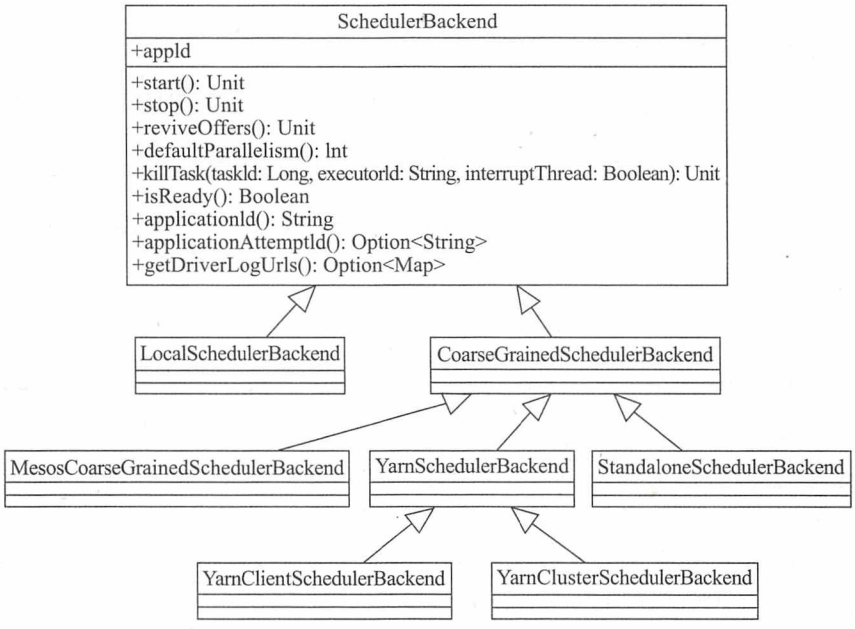


图 7-14　SchedulerBackend 的继承体系

从图 7-14 中可以看到 SchedulerBackend 有两个子类，分别如下。

1）CoarseGrainedSchedulerBackend：等待 CoarseGrainedExecutorBackend 进行连接的 SchedulerBackend 实现。由 CoarseGrainedSchedulerBackend 建立的 CoarseGrainedExecutor-Backend 进程将会一直存在，真正的 Executor 线程将在 Coarse GrainedExecutor-Backend 进程中执行。

2）LocalSchedulerBackend：local 模式中的调度后端接口。在 local 模式下，Executor、LocalSchedulerBackend、Driver 都运行在同一个 JVM 进程中。

CoarseGrainedSchedulerBackend 的子类 StandaloneSchedulerBackend 是 Standalone 部署模式下的 SchedulerBackend 实现。对于 Mesos 或 YAM，分别实现了继承自 CoarseGrained Scheduler Backend 的 MesosCoarseGrainedSchedulerBackend 和 YarnScheduler Backend。YarnSchedulerBackend 还有 YarnClientSchedulerBackend 和 YarnCluster SchedulerBackend 两个子类。

7.8.2　LocalSchedulerBackend 的实现分析

我们虽然对特质 SchedulerBackend 的定义及继承体系有所了解，但是我们依然不是很

明白 SchedulerBackend 的子类该如何实现。这里以 SchedulerBackend 在 local 部署模式下的实现类 LocalSchedulerBackend 为例，来详细分析其实现内容。

1. LocalEndpoint 的实现分析

LocalSchedulerBackend 与其他组件的通信都依赖于 LocalEndpoint，所以我们需要先来认识它。LocalEndpoint 包含以下属性。

- rpcEnv：即 RpcEnv。
- userClassPath：用户指定的 ClassPath。
- scheduler：即 Driver 中的 TaskSchedulerImpl。
- executorBackend：与 LocalEndpoint 相关联的 LocalSchedulerBackend。
- totalCores：用于执行任务的 CPU 内核总数。local 模式下，totalCores 固定为 1。
- freeCores：空闲的 CPU 内核数。应用程序提交的 Task 正式运行之前，freeCores 与 totalCores 相等。
- localExecutorId：local 部署模式下，与 Driver 处于同一 JVM 进程的 Executor 的身份标识。由于 LocalEndpoint 只在 local 模式中使用，因此 localExecutorId 固定为 driver。
- localExecutorHostname：与 Driver 处于同一 JVM 进程的 Executor 所在的 Host。由于 LocalEndpoint 只在 local 模式中使用，因此 localExecutorHostname 固定为 localhost。
- executor：与 Driver 处于同一 JVM 进程的 Executor。由于 LocalEndpoint 的 totalCores 等于 1，因此应用本地有且只有一个 Executor，且此 Executor 在 LocalEndpoint 构造的过程中就已经实例化。

LocalEndpoint 其实现如代码清单 7-88 所示。重写了 RpcEndpoint 的 receive 方法和 receiveAndReply 方法。

代码清单7-88 LocalEndpoint接收的消息实现

```
override def receive: PartialFunction[Any, Unit] = {
  case ReviveOffers =>
    reviveOffers()
  case StatusUpdate(taskId, state, serializedData) =>
    scheduler.statusUpdate(taskId, state, serializedData)
    if (TaskState.isFinished(state)) {
      freeCores += scheduler.CPUS_PER_TASK
      reviveOffers()
    }
  case KillTask(taskId, interruptThread) =>
    executor.killTask(taskId, interruptThread)
}

override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
  case StopExecutor =>
    executor.stop()
    context.reply(true)
}
```

receive 方法在处理 ReviveOffers 和 StatusUpdate 消息时，都会调用 reviveOffers（见代码清单 7-89）方法给 Task 分配资源。

代码清单7-89 LocalEndpoint的reviveOffers

```
def reviveOffers() {
  val offers = IndexedSeq(new WorkerOffer(localExecutorId, localExecutorHostname,
       freeCores))
  for (task <- scheduler.resourceOffers(offers).flatten) { // 给Task分配资源
    freeCores -= scheduler.CPUS_PER_TASK // 将空闲CPU内核数freeCores减1
    executor.launchTask(executorBackend, taskId = task.taskId, attemptNumber = task.
        attemptNumber,
      task.name, task.serializedTask) // 运行Task
  }
}
```

根据代码清单 7-89，LocalEndpoint 的 reviveOffers 方法的执行步骤如下。

1）创建只包含一个 WorkerOffer（理解为“就业机会”似乎更加生动，由于 totalCores 为 1，因此只有一个职位）的序列。样例类 WorkerOffer 的实现如下。

```
private[spark] case class WorkerOffer(executorId: String, host: String, cores: Int)
```

根据对 LocalEndpoint 的分析，此处 WorkerOffer 的 executorId 为 driver，host 为 localhost，cores 为 1。

2）调用 TaskSchedulerImpl 的 resourceOffers 方法（见代码清单 7-101）给 Task 分配资源。

3）将空闲 CPU 内核数 freeCores 减 1。

4）调用 Executor 的 launchTask 方法运行 Task。Executor 将在第 9 章详细介绍。

2. LocalSchedulerBackend 的实现

我们从了解 LocalSchedulerBackend 的属性开始，逐步分析 LocalSchedulerBackend 的实现。LocalSchedulerBackend 的属性如下。

- conf：即 SparkConf。
- scheduler：即 TaskSchedulerImpl。
- totalCores：LocalSchedulerBackend 的 CPU 内核数，固定为 1。
- appId：当前应用程序的身份标识。local 模式下的 appId 以 local- 为前缀，以系统当前时间戳为后缀，local 模式下应用程序的身份标识是在构造 LocalSchedulerBackend 的时候生成的。
- localEndpoint：即 LocalEndpoint 的 NettyRpcEndpointRef。
- userClassPath：用户指定的类路径。可以通过 spark.executor.extraClassPath 属性进行配置，配置时可以用英文逗号分隔多个类路径。
- listenerBus：即 SparkContext 中创建的 LiveListenerBus。
- launcherBackend：LauncherBackend 的匿名实现类的实例。此匿名实现类实现了 LauncherBackend 的 onStopRequest 方法（见代码清单 7-90），用于停止 Executor、将

launcherBackend 的状态标记为 KILLED、关闭 launcherBackend 与 LauncherServer 之间的 Socket 连接。

代码清单7-90 LauncherBackend的匿名实现类

```
private val launcherBackend = new LauncherBackend() {
  override def onStopRequest(): Unit = stop(SparkAppHandle.State.KILLED)
}
```

在构造 LocalSchedulerBackend 的最后，会调用 launcherBackend 的 connect 方法（见代码清单 7-83）与 LauncherServer 进行连接。

```
launcherBackend.connect()
```

最后来看看 LocalSchedulerBackend 提供的常用方法。

- start：启动 LocalSchedulerBackend。LocalSchedulerBackend 的 start 方法的实现如代码清单 7-91 所示。

代码清单7-91 启动LocalSchedulerBackend

```
override def start() {
  val rpcEnv = SparkEnv.get.rpcEnv
  val executorEndpoint = new LocalEndpoint(rpcEnv, userClassPath, scheduler, this,
      totalCores)
  localEndpoint = rpcEnv.setupEndpoint("LocalSchedulerBackendEndpoint", executor
      Endpoint)
  listenerBus.post(SparkListenerExecutorAdded(
    System.currentTimeMillis,
    executorEndpoint.localExecutorId,
    new ExecutorInfo(executorEndpoint.localExecutorHostname, totalCores, Map.empty)))
  launcherBackend.setAppId(appId) // 向LauncherServer发送SetAppId消息
  launcherBackend.setState(SparkAppHandle.State.RUNNING) // 向LauncherServer发送
      SetState消息
}
```

根据代码清单 7-91，LocalSchedulerBackend 的 start 方法的执行步骤如下。

1）创建 LocalEndpoint 并注册到 RpcEnv 中，然后由 localEndpoint 属性持有 LocalEndpoint 的 NettyRpcEndpointRef。

2）向 LiveListenerBus 投递 SparkListenerExecutorAdded 事件。local 模式下 SparkListener ExecutorAdded 事件携带的 time 为系统当前时间，executorId 为 driver，executorInfo（即 ExecutorInfo，ExecutorInfo 的 executorHost 属性为 localhost，totalCores 为 1，logUrlMap 为空）。

3）调用 LauncherBackend 的 setAppId 方法向 LauncherServer 发送 SetAppId 消息。

4）调用 LauncherBackend 的 setState 方法向 LauncherServer 发送 SetState 消息。

- reviveOffers：对 Task 进行资源分配后运行 Task。LocalSchedulerBackend 的 reviveOffers 方法的实现如代码清单 7-92 所示。

代码清单7-92 LocalSchedulerBackend的reviveOffers方法

```
override def reviveOffers() {
  localEndpoint.send(ReviveOffers)
}
```

根据代码清单 7-92，LocalSchedulerBackend 的 reviveOffers 方法将向 LocalEndpoint 发送 ReviveOffers 消息。LocalEndpoint 接收 ReviveOffers 消息后（见代码清单 7-88），会调用 reviveOffers 方法给下一个要调度的 Task 分配资源并运行 Task。

- statusUpdate：Task 的状态更新。LocalSchedulerBackend 实现了特质 ExecutorBackend 的唯一方法 statusUpdate，其实现如代码清单 7-93 所示。

代码清单7-93 Task的状态更新

```
override def statusUpdate(taskId: Long, state: TaskState, serializedData: ByteBuffer) {
  localEndpoint.send(StatusUpdate(taskId, state, serializedData))
}
```

根据代码清单 7-93，LocalSchedulerBackend 的 statusUpdate 方法将向 LocalEndpoint 发送 StatusUpdate 消息。LocalEndpoint 接收到 StatusUpdate 消息后（见代码清单 7-88），会首先调用 TaskSchedulerImpl 的 statusUpdate 方法（见代码清单 7-103）更新 Task 状态。

7.9 任务结果获取器 TaskResultGetter

TaskResultGetter 用于对序列化的 Task 执行结果进行反序列化，以得到 Task 执行结果。TaskResultGetter 也可远程获取 Task 执行结果。

TaskResultGetter 包含以下属性。

- sparkEnv：即 SparkEnv。
- scheduler：即 TaskSchedulerImpl。
- THREADS：获取 Task 执行结果的线程数。可通过 spark.resultGetter.threads 属性配置，默认为 4。
- getTaskResultExecutor：使用 Executors 的 newFixedThreadPool 方法创建的 ThreadPool Executor，用于提交获取 Task 执行结果的线程。线程池的大小由 THREADS 决定。
- serializer：类型为 ThreadLocal[SerializerInstance]，通过使用本地线程缓存，保证在使用 SerializerInstance 时是线程安全的。
- taskResultSerializer：类型为 ThreadLocal[SerializerInstance]，通过使用本地线程缓存，保证在使用 SerializerInstance 对 Task 的执行结果进行反序列化时是线程安全的。

7.9.1 处理成功的 Task

TaskResultGetter 的 enqueueSuccessfulTask 方法用于处理执行成功的 Task 的执行结果，

其实现如代码清单 7-94 所示。

代码清单7-94 enqueueSuccessfulTask方法

```
def enqueueSuccessfulTask(
    taskSetManager: TaskSetManager,
    tid: Long,
    serializedData: ByteBuffer): Unit = {
  getTaskResultExecutor.execute(new Runnable {
    override def run(): Unit = Utils.logUncaughtExceptions {
      try { // 对Task的执行结果反序列化
        val (result, size) = serializer.get().deserialize[TaskResult[_]](serialized
            Data) match {
          case directResult: DirectTaskResult[_] =>
            if (!taskSetManager.canFetchMoreResults(serializedData.limit())) {
              return
            }
            directResult.value(taskResultSerializer.get()) // 获取Task执行结果
            (directResult, serializedData.limit())
          case IndirectTaskResult(blockId, size) =>
            if (!taskSetManager.canFetchMoreResults(size)) {
              sparkEnv.blockManager.master.removeBlock(blockId)
              return
            }
            logDebug("Fetching indirect task result for TID %s".format(tid))
            scheduler.handleTaskGettingResult(taskSetManager, tid)
            val serializedTaskResult = sparkEnv.blockManager.getRemoteBytes
                (blockId) // 下载结果
            if (!serializedTaskResult.isDefined) {
              scheduler.handleFailedTask(
                taskSetManager, tid, TaskState.FINISHED, TaskResultLost)
              return
            }
            val deserializedResult = serializer.get().deserialize[DirectTask
                Result[_]](
              // 对下载到的数据反序列化得到Task的执行结果
              serializedTaskResult.get.toByteBuffer)
            deserializedResult.value(taskResultSerializer.get())
            sparkEnv.blockManager.master.removeBlock(blockId)
            (deserializedResult, size)
        }

        result.accumUpdates = result.accumUpdates.map { a =>
          if (a.name == Some(InternalAccumulator.RESULT_SIZE)) {
            val acc = a.asInstanceOf[LongAccumulator]
            assert(acc.sum == 0L, "task result size should not have been set on the
                executors")
            acc.setValue(size.toLong)
            acc
          } else {
            a
          }
        }
        scheduler.handleSuccessfulTask(taskSetManager, tid, result) //
      } catch {
        // 忽略异常处理代码.
```

```
        }
      }
    })
  }
```

根据代码清单 7-94，enqueueSuccessfulTask 方法实际向 getTaskResultExecutor 提交了一个获取 Task 执行结果的任务，这个任务的执行步骤如下。

1）对 Task 的执行结果反序列化，如果 Task 的结果类型为 DirectTaskResult，说明 Task 的执行结果保存在 DirectTaskResult 中，此时只需要对 DirectTaskResult 保存的数据（即 DirectTaskResult 的 valueBytes 属性）进行反序列化就可以得到。

2）对 Task 的执行结果反序列化，如果 Task 的结果类型为 IndirectTaskResult，说明 Task 的执行结果没有保存在 IndirectTaskResult 中，此时需要调用 TaskSchedulerImpl 的 handleTaskGettingResult 方法（此方法的实现非常简单，这里就不过多介绍了），向 DAGSchedulerEventProcessLoop 投递 GettingResultEvent 事件，然后调用 BlockManager 的 getRemoteBytes 方法（见代码清单 6-75），从运行 Task 的节点上下载 Block，最后对下载到的数据反序列化得到 Task 的执行结果。

3）更新 Task 的执行结果的累加器中的结果大小。

4）调用 TaskSchedulerImpl 的 handleSuccessfulTask 方法（见代码清单 7-105）。



虽然 enqueueSuccessfulTask 方法最终可以得到 DirectTaskResult，但是 DirectTaskResult 并不像它的字面意思一样代表 Task 的执行结果，对于 ResultTask 来说，DirectTaskResult 的 value 的确是 Task 的执行结果，但对于 ShuffleMapTask 而言，实际是任务的状态。

7.9.2 处理失败的 Task

TaskResultGetter 的 enqueueFailedTask 方法用于处理执行失败的 Task 的执行结果，其实现如代码清单 7-95 所示。

代码清单7-95 enqueueFailedTask方法

```
def enqueueFailedTask(taskSetManager: TaskSetManager, tid: Long, taskState: TaskState,
  serializedData: ByteBuffer) {
  var reason : TaskFailedReason = UnknownReason
  try {
    getTaskResultExecutor.execute(new Runnable {
      override def run(): Unit = Utils.logUncaughtExceptions {
        val loader = Utils.getContextOrSparkClassLoader
        try {
          if (serializedData != null && serializedData.limit() > 0) {
            reason = serializer.get().deserialize[TaskFailedReason](
              serializedData, loader) // 对执行结果反序列化，得到失败原因
          }
        } catch {
```

```
            case cnd: ClassNotFoundException =>
              logError(
                "Could not deserialize TaskEndReason: ClassNotFound with classloader
                    " + loader)
            case ex: Exception => // No-op
          }
          scheduler.handleFailedTask(taskSetManager, tid, taskState, reason) // 处理
            失败的Task
        }
      })
    } catch {
      case e: RejectedExecutionException if sparkEnv.isStopped =>
        // ignore it
    }
  }
```

根据代码清单7-95，enqueueFailedTask方法实际向getTaskResultExecutor提交了一个获取Task执行结果的任务，这个任务的执行步骤如下。

1）对执行结果反序列化，得到类型为TaskFailedReason的失败原因。

2）调用TaskSchedulerImpl的handleFailedTask方法将失败的Task重新放入待处理的Task列表，并通知DAGScheduler重新调度。TaskSchedulerImpl的handleFailedTask方法的分析方式与handleSuccessfulTask方法类似，留给读者去分析。

7.10 任务调度器TaskScheduler

任务调度器TaskScheduler定义了对任务进行调度的接口规范，允许向Spark调度系统插入不同的TaskScheduler实现，但目前只有TaskSchedulerImpl这一个具体实现。TaskScheduler只为单个Driver调度任务。TaskSchedulerImpl的功能包括接收DAGScheduler给每个Stage创建的Task集合，按照调度算法将资源分配给Task，将Task交给Spark集群不同节点上的Executor运行，在这些Task执行失败时进行重试，通过推断执行减轻落后的Task对整体作业进度的影响。

Spark的资源调度分为两层：

第一层是Cluster Manager（在YARN模式下为Resource Manager，在Mesos模式下为Mesos Master，在Standalone模式下为Master）将资源分配给Application；

第二层是Application进一步将资源分配给Application的各个Task。TaskSchedulerImpl中的资源调度就是第二层的资源调度。

7.10.1 TaskSchedulerImpl的属性

TaskSchedulerImpl中有很多成员属性，理解这些属性的含义是深入TaskSchedulerImpl的前提。TaskSchedulerImpl的成员属性如下。

- sc：此属性持有SparkContext的引用。

- maxTaskFailures：任务失败的最大次数。
- isLocal：是否是 Local 部署模式。Local 部署模式下的 LocalSchedulerBackend 已在 7.8.2 节介绍，更多部署模式的内容将在第 9 章中详细介绍。
- conf：即 SparkConf。
- SPECULATION_INTERVAL_MS：任务推断执行的时间间隔。可以通过 spark.speculation.interval 属性进行配置，默认为 100ms。
- MIN_TIME_TO_SPECULATION：用于保证原始任务至少需要运行的时间。原始任务只有超过此时间限制，才允许启动副本任务。这可以避免原始任务执行太短的时间就被推断执行副本任务。MIN_TIME_TO_SPECULATION 的大小固定为 100。
- speculationScheduler：对任务调度进行推断执行的 ScheduledThreadPoolExecutor。由 speculationScheduler 创建的线程以 task-scheduler-speculation 为前缀。
- STARVATION_TIMEOUT_MS：判断 TaskSet 饥饿的阈值。可通过 spark.starvation.timeout 属性配置，默认为 15s。
- CPUS_PER_TASK：每个 Task 需要分配的 CPU 核数。可通过 spark.task.cpus 属性配置，默认为 1。
- taskSetsByStageIdAndAttempt：数据类型为 HashMap[Int, HashMap[Int, TaskSetManager]]，是用于 StageId、Attempt、TaskSetManager 的二级缓存。
- taskIdToTaskSetManager：Task 与所属 TaskSetManager 的映射关系。
- taskIdToExecutorId：Task 与执行此 Task 的 Executor 之间的映射关系。
- hasReceivedTask：标记 TaskSchedulerImpl 是否已经接收到 Task。
- hasLaunchedTask：标记 TaskSchedulerImpl 接收的 Task 是否已经有运行过的。
- starvationTimer：处理饥饿的定时器。
- nextTaskId：类型为 AtomicLong，用于生成新提交 Task 的标识。
- executorIdToRunningTaskIds：类型为 HashMap[String, HashSet[Long]]，用于缓存 Executor 与运行在此 Executor 上的任务之间的映射关系，由此看出一个 Executor 上可以运行多个 Task。
- hostToExecutors：类型为 HashMap[String, HashSet[String]]，用于缓存机器的 Host 与运行在此机器上的 Executor 之间的映射关系，由此可以看出机器与 Executor 之间是一对多的关系。
- hostsByRack：类型为 HashMap[String, HashSet[String]]，用于缓存机器所在的机架与机架上机器的 Host 之间的映射关系，由此可以看出机架与机器之间是一对多的关系。
- executorIdToHost：Executor 与 Executor 运行所在机器的 Host 之间的映射关系。
- dagScheduler：即 DAGScheduler。
- backend：即调度后端接口 SchedulerBackend。
- mapOutputTracker：即 SparkEnv 的子组件 MapOutputTrackerMaster。
- schedulableBuilder：调度池构建器，即 SchedulableBuilder。

- rootPool：根调度池，类型为 Pool。
- schedulingModeConf：调度模式配置。可以通过 spark.scheduler.mode 属性配置，默认为 FIFO。
- schedulingMode：调度模式。此属性依据 schedulingModeConf 获取枚举类型 SchedulingMode 的具体值。SchedulingMode 共有 FAIR、FIFO、NONE 三种枚举值。
- taskResultGetter：类型为 TaskResultGetter，它的作用是通过线程池（此线程池由 Executors.newFixedThreadPool 创建，大小默认为 4，生成的线程名以 task-result-getter 开头），对 Slave 发送的 Task 的执行结果进行处理。

7.10.2 TaskSchedulerImpl 的初始化

在 4.5 节介绍创建任务调度器的时候，在代码清单 4-18 中展示了针对不同 master 配置，分别以不同方式创建 TaskSchedulerImpl 的内容。无论何种方式创建 TaskSchedulerImpl，要想让 TaskSchedulerImpl 发挥作用，必须都调用 TaskSchedulerImpl 的 initialize 方法（见代码清单 7-96）对 TaskSchedulerImpl 进行初始化。

代码清单7-96 TaskSchedulerImpl的初始化

```
def initialize(backend: SchedulerBackend) {
  this.backend = backend
  rootPool = new Pool("", schedulingMode, 0, 0) // 创建根调度池
  schedulableBuilder = { // 根据调度模式，创建相应的调度池构建器
    schedulingMode match {
      case SchedulingMode.FIFO =>
        new FIFOSchedulableBuilder(rootPool)
      case SchedulingMode.FAIR =>
        new FairSchedulableBuilder(rootPool, conf)
      case _ =>
        throw new IllegalArgumentException(s"Unsupported spark.scheduler.mode:
            $schedulingMode")
    }
  }
  schedulableBuilder.buildPools() // 构建调度池
}
```

根据代码清单 7-96，initialize 方法的执行步骤如下。

1）使用参数传递的 SchedulerBackend 设置 TaskSchedulerImpl 的 backend 属性。

2）创建根调度池。

3）根据调度模式，创建相应的调度池构建器。由于 SchedulingMode 默认为 FIFO，所以创建的调度构建器默认为 FIFOSchedulableBuilder。

4）调用调度池构建器的 buildPools 方法构建调度池。

7.10.3 TaskSchedulerImpl 的启动

根据 4.5 节的内容，我们知道启动任务调度器是通过调用其 start 方法实现的。Task

SchedulerImpl 的 start 方法的实现如代码清单 7-97 所示。

代码清单7-97 TaskSchedulerImpl的启动

```
override def start() {
  backend.start() // 启动SchedulerBackend
  if (!isLocal && conf.getBoolean("spark.speculation", false)) { // 设置检查可推断任
                                                                 // 务的定时器
    logInfo("Starting speculative execution thread")
    speculationScheduler.scheduleAtFixedRate(new Runnable {
      override def run(): Unit = Utils.tryOrStopSparkContext(sc) {
        checkSpeculatableTasks()
      }
    }, SPECULATION_INTERVAL_MS, SPECULATION_INTERVAL_MS, TimeUnit.MILLISECONDS)
  }
}
```

根据代码清单 7-97，start 方法的执行步骤如下。

1）调用 SchedulerBackend 的 start 方法启动 SchedulerBackend。

2）当应用不是在 Local 模式下，并且设置了推断执行（即 spark.speculation 属性为 true），那么设置一个执行间隔为 SPECULATION_INTERVAL_MS（默认为 100ms）的检查可推断任务的定时器。此定时器通过调用 checkSpeculatableTasks 方法（见代码清单 7-98）来检查可推断任务。

代码清单7-98 检测可推断执行任务

```
def checkSpeculatableTasks() {
  var shouldRevive = false
  synchronized {
    shouldRevive = rootPool.checkSpeculatableTasks(MIN_TIME_TO_SPECULATION)
  }
  if (shouldRevive) {
    backend.reviveOffers()
  }
}
```

根据代码清单 7-98，checkSpeculatableTasks 方法实际依赖于 rootPool 的 checkSpeculatable Tasks 方法（见代码清单 7-52）。如果检查到有可以推断执行的任务，则调用 Scheduler Backend 的 reviveOffers 方法。以 Local 模式下的 LocalSchedulerBackend 为例，LocalScheduler Backend 的 reviveOffers 方法（见代码清单 7-92）将向 LocalEndpoint 发送 ReviveOffers 消息，LocalEndpoint 接收到 ReviveOffers 消息后（见代码清单 7-88），将调用 LocalEndpoint 的 reviveOffers 方法（见代码清单 7-89）分配资源并运行 Task。

7.10.4 TaskSchedulerImpl 与 Task 的提交

DAGScheduler 将 Stage 中各个分区的 Task 封装为 TaskSet 后，会将 TaskSet 交给 Task SchedulerImpl 处理。TaskSchedulerImpl 的 submitTasks 方法是这一过程的入口，其实现如

代码清单 7-99 所示。

代码清单7-99 提交Task

```
override def submitTasks(taskSet: TaskSet) {
  val tasks = taskSet.tasks // 获取TaskSet中的所有Task
  logInfo("Adding task set " + taskSet.id + " with " + tasks.length + " tasks")
  this.synchronized {
    val manager = createTaskSetManager(taskSet, maxTaskFailures) // 创建TaskSetManager
    val stage = taskSet.stageId
    val stageTaskSets =
      taskSetsByStageIdAndAttempt.getOrElseUpdate(stage, new HashMap[Int, TaskSet
          Manager])
    stageTaskSets(taskSet.stageAttemptId) = manager
    val conflictingTaskSet = stageTaskSets.exists { case (_, ts) =>
      ts.taskSet != taskSet && !ts.isZombie
    }
    if (conflictingTaskSet) {
      throw new IllegalStateException(s"more than one active taskSet for stage
          $stage:" +
        s" ${stageTaskSets.toSeq.map{_._2.taskSet.id}.mkString(",")}")
    }
    schedulableBuilder.addTaskSetManager(manager, manager.taskSet.properties)

    if (!isLocal && !hasReceivedTask) { // 设置检查TaskSchedulerImpl的饥饿状况的定时器
      starvationTimer.scheduleAtFixedRate(new TimerTask() {
        override def run() {
          if (!hasLaunchedTask) {
            logWarning("Initial job has not accepted any resources; " +
              "check your cluster UI to ensure that workers are registered " +
              "and have sufficient resources")
          } else {
            this.cancel()
          }
        }
      }, STARVATION_TIMEOUT_MS, STARVATION_TIMEOUT_MS)
    }
    hasReceivedTask = true // 表示TaskSchedulerImpl已经接收到Task
  }
  backend.reviveOffers() // 给Task分配资源并运行Task
}
```

根据代码清单 7-99，submitTasks 方法的执行步骤如下。

1）获取 TaskSet 中的所有 Task。

2）调用 createTaskSetManager 方法（见代码清单 7-100）创建 TaskSetManager。

3）在 taskSetsByStageIdAndAttempt 中设置 TaskSet 关联的 Stage、Stage 尝试及刚创建的 TaskSetManager 之间的三级映射关系。

4）对当前 TaskSet 进行冲突检测，即 taskSetsByStageIdAndAttempt 中不应该存在同属于当前 Stage，但是 TaskSet 却不相同的情况。

5）调用调度池构建器的 addTaskSetManager 方法（见代码清单 7-56 或代码清单 7-60），将刚创建的 TaskSetManager 添加到调度池构建器的调度池中。

6）如果当前应用程序不是 Local 模式并且 TaskSchedulerImpl 还没有接收到 Task，那么设置一个定时器按照 STARVATION_TIMEOUT_MS 指定的时间间隔检查 TaskSchedulerImpl 的饥饿状况，当 TaskSchedulerImpl 已经运行 Task 后，取消此定时器。

7）将 hasReceivedTask 设置为 true，以表示 TaskSchedulerImpl 已经接收到 Task。

8）调用 SchedulerBackend 的 reviveOffers 方法给 Task 分配资源并运行 Task。

代码清单7-100 创建TaskSetManager

```
private[scheduler] def createTaskSetManager(
    taskSet: TaskSet,
    maxTaskFailures: Int): TaskSetManager = {
  new TaskSetManager(this, taskSet, maxTaskFailures)
}
```

7.10.5 TaskSchedulerImpl 与资源分配

以 Local 模式下 SchedulerBackend 的实现 LocalSchedulerBackend 为例，LocalSchedulerBackend 的 reviveOffers 方法（见代码清单 7-92）实际向 LocalEndpoint 发送了 ReviveOffers 消息。LocalEndpoint 接收到 ReviveOffers 消息后，将调用 LocalEndpoint 自己的 reviveOffers 方法（见代码清单 7-89）。根据之前对 LocalEndpoint 的 reviveOffers 方法的介绍，最终将调用 TaskSchedulerImpl 的 resourceOffers 方法（见代码清单 7-101）给 Task 分配资源。

代码清单7-101 给Task分配资源

```
def resourceOffers(offers: IndexedSeq[WorkerOffer]): Seq[Seq[TaskDescription]] =
    synchronized {
  var newExecAvail = false
  for (o <- offers) {
    if (!hostToExecutors.contains(o.host)) {
      hostToExecutors(o.host) = new HashSet[String]()
    }
    if (!executorIdToRunningTaskIds.contains(o.executorId)) { // 更新Host与Executor
                                                              // 的各种映射关系
      hostToExecutors(o.host) += o.executorId
      executorAdded(o.executorId, o.host)
      executorIdToHost(o.executorId) = o.host
      executorIdToRunningTaskIds(o.executorId) = HashSet[Long]()
      newExecAvail = true // 标记添加了新的Executor
    }
    for (rack <- getRackForHost(o.host)) { // 更新Host与机架之间的关系
      hostsByRack.getOrElseUpdate(rack, new HashSet[String]()) += o.host
    }
  }

  val shuffledOffers = Random.shuffle(offers) // 随机洗牌，避免将任务总是分配给同样一组Worker
  val tasks = shuffledOffers.map(o => new ArrayBuffer[TaskDescription](o.cores))
  // 统计每个Worker的可用的CPU核数
  val availableCpus = shuffledOffers.map(o => o.cores).toArray
  // 所有TaskSetManager按照调度算法排序
  val sortedTaskSets = rootPool.getSortedTaskSetQueue
  for (taskSet <- sortedTaskSets) {
```

```
      logDebug("parentName: %s, name: %s, runningTasks: %s".format(
        taskSet.parent.name, taskSet.name, taskSet.runningTasks))
      if (newExecAvail) {
        taskSet.executorAdded() // 重新计算TaskSet的本地性
      }
    }

    for (taskSet <- sortedTaskSets) {
      var launchedAnyTask = false
      var launchedTaskAtCurrentMaxLocality = false
      for (currentMaxLocality <- taskSet.myLocalityLevels) { //按照最大本地性的原则,
                                                             //给Task提供资源
        do {
          launchedTaskAtCurrentMaxLocality = resourceOfferSingleTaskSet(
            taskSet, currentMaxLocality, shuffledOffers, availableCpus, tasks)
          launchedAnyTask |= launchedTaskAtCurrentMaxLocality
        } while (launchedTaskAtCurrentMaxLocality)
      }
      if (!launchedAnyTask) {
        taskSet.abortIfCompletelyBlacklisted(hostToExecutors)
      }
    }

    if (tasks.size > 0) {
      hasLaunchedTask = true
    }
    return tasks // 返回已经获得了资源的任务列表
  }
```

根据代码清单 7-101，resourceOffers 方法的执行步骤如下。

1）遍历 WorkerOffer 序列，对每一个 WorkerOffer 执行以下操作。

① 更新 Host 与 Executor 的各种映射关系。

② 调用 TaskSchedulerImpl 的 executorAdded 方法（此方法实际仅仅调用了代码清单 7-30 所示的 DAGScheduler 的 executorAdded 方法）向 DAGScheduler 的 DAGScheduler EventProcessLoop 投递 ExecutorAdded 事件。

③ 标记添加了新的 Executor（即将 newExecAvail 设置为 true）。

④ 更新 Host 与机架之间的关系。

小贴士：这里的 hostToExecutors 及 hostsByRack 是为了在资源分配时计算 Task 本地性时使用。

2）对所有 WorkerOffer 随机洗牌，避免将任务总是分配给同样一组 Worker。

3）根据每个 WorkerOffer 的可用的 CPU 核数创建同等尺寸的任务描述（TaskDescription）数组。

4）将每个 WorkerOffer 的可用的 CPU 核数统计到可用 CPU（availableCpus）数组中。

5）调用 rootPool 的 getSortedTaskSetQueue 方法（见代码清单 7-53），对 rootPool 中的所有 TaskSetManager 按照调度算法排序。

6）如果 newExecAvail 为 true，那么调用每个 TaskSetManager 的 executorAdded 方法。此 executorAdded 方法实际调用了 computeValidLocalityLevels 方法（见代码清单 7-64）重新计算 TaskSet 的本地性。

7）遍历 TaskSetManager，按照最大本地性的原则（即从高本地性级别到低本地性级别）调用 resourceOfferSingleTaskSet 方法（见代码清单 7-102），给单个 TaskSet 中的 Task 提供资源。如果在任何 TaskSet 所允许的本地性级别下，TaskSet 中没有任何一个任务获得了资源，那么将调用 TaskSetManager 的 abortIfCompletelyBlacklisted 方法（由于 abortIfCompletely Blacklisted 方法与调度流程的主逻辑相离较远，所以留给感兴趣的读者自行研究），放弃在黑名单中的 Task。

8）返回生成的 TaskDescription 列表，即已经获得了资源的任务列表。

代码清单7-102 给单个TaskSet中的Task提供资源

```
private def resourceOfferSingleTaskSet(
    taskSet: TaskSetManager,
    maxLocality: TaskLocality,
    shuffledOffers: Seq[WorkerOffer],
    availableCpus: Array[Int],
    tasks: IndexedSeq[ArrayBuffer[TaskDescription]]) : Boolean = {
  var launchedTask = false
  for (i <- 0 until shuffledOffers.size) {
    val execId = shuffledOffers(i).executorId
    val host = shuffledOffers(i).host
    if (availableCpus(i) >= CPUS_PER_TASK) {
      try { // 给符合条件的待处理Task创建TaskDescription
        for (task <- taskSet.resourceOffer(execId, host, maxLocality)) {
          tasks(i) += task
          val tid = task.taskId
          taskIdToTaskSetManager(tid) = taskSet
          taskIdToExecutorId(tid) = execId
          executorIdToRunningTaskIds(execId).add(tid)
          availableCpus(i) -= CPUS_PER_TASK
          assert(availableCpus(i) >= 0)
          launchedTask = true
        }
      } catch {
        case e: TaskNotSerializableException =>
            logError(s"Resource offer failed, task set ${taskSet.name} was not
serializable")
          return launchedTask
      }
    }
  }
  return launchedTask
}
```

根据代码清单 7-102，resourceOfferSingleTaskSet 方法将遍历 WorkerOffer 序列，对每一个 WorkerOffer 执行以下操作。

1）获取 WorkerOffer 的 Executor 的身份标识。

2）获取 WorkerOffer 的 Host。

3）如果 WorkerOffer 的可用的 CPU 核数大于等于 CPUS_PER_TASK，则执行以下操作。

① 调用 TaskSetManager 的 resourceOffer 方法（见代码清单 7-73），给符合条件的待处理 Task 创建 TaskDescription。

② 将 TaskDescription 添加到 tasks 数组。

③ 更新 Task 的身份标识与 TaskSet、Executor 的身份标识相关的缓存映射。

④ 由于给 Task 分配了 CPUS_PER_TASK 指定数量的 CPU 内核数，因此 WorkerOffer 的可用的 CPU 核数减去 CPUS_PER_TASK。

⑤ 返回 launchedTask，即是否已经给 TaskSet 中的某个 Task 分配到了资源。

7.10.6 TaskSchedulerImpl 的调度流程

本节通过对 TaskSchedulerImpl 的属性、初始化、启动、提交 Task、给 Task 分配资源等内容的分析，想必读者对 TaskSchedulerImpl 的调度流程有了深入的理解，现在可以用图 7-15 来表示 TaskSchedulerImpl 的调度流程。

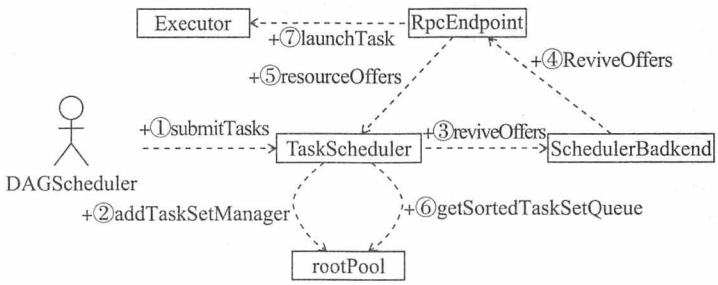


图 7-15 TaskSchedulerImpl 的调度流程

图 7-15 从抽象的角度描绘了 TaskSchedulerImpl 的调度流程，使用了 SchedulerBackend，而不是 SchedulerBackend 的具体实现（如 LocalSchedulerBackend），RpcEndpoint 代表 SchedulerBackend 的具体实现中与其他组件进行通信的实例（如 LocalSchedulerBackend 中的 LocalEndpoint）。这里对图 7-15 中的记号进行说明。

记号①：代表 DAGScheduler 调用 TaskScheduler 的 submitTasks 方法向 TaskScheduler 提交 TaskSet。

记号②：代表 TaskScheduler 接收到 TaskSet 后，创建对此 TaskSet 进行管理的 TaskSet Manager，并将此 TaskSetManager 通过调度池构建器添加到根调度池中。

记号③：代表 TaskScheduler 调用 SchedulerBackend 的 reviveOffers 方法给 Task 提供资源。

记号④：SchedulerBackend 向 RpcEndpoint 发送 ReviveOffers 消息。

记号⑤：RpcEndpoint 将调用 TaskScheduler 的 resourceOffers 方法给 Task 提供资源。

仅供非商业用途或交流学习使用

记号⑥：TaskScheduler 调用根调度池的 getSortedTaskSetQueue 方法对所有 TaskSetManager 按照调度算法进行排序后，对 TaskSetManager 管理的 TaskSet 按照“最大本地性”的原则选择其中的 Task，最后为 Task 创建尝试执行信息、对 Task 进行序列化、生成 TaskDescription 等。

记号⑦：调用 Executor 的 launchTask 方法运行 Task 尝试。

7.10.7 TaskSchedulerImpl 对执行结果的处理

Task 在执行的时候会不断发送 StatusUpdate 消息，在 Local 模式下，LocalEndpoint 接收到 StatusUpdate 消息后（见代码清单 7-88）会先匹配执行 TaskSchedulerImpl 的 statusUpdate 方法，然后调用 reviveOffers 方法（见代码清单 7-89）给其他 Task 分配资源。

TaskSchedulerImpl 的 statusUpdate 方法（代码清单 7-103）用于更新 Task 的状态。Task 的状态包括：运行中（RUNNING）、已完成（FINISHED）、失败（FAILED）、被杀死（KILLED）、丢失（LOST）五种。会从 taskIdToTaskSetId、taskIdToExecutorId 中移除此任务，并且调用 taskResultGetter 的 enqueueSuccessfulTask 方法。

代码清单7-103 更新Task的状态

```
def statusUpdate(tid: Long, state: TaskState, serializedData: ByteBuffer) {
  var failedExecutor: Option[String] = None
  var reason: Option[ExecutorLossReason] = None
  synchronized {
    try {
      taskIdToTaskSetManager.get(tid) match {
        case Some(taskSet) =>
          if (state == TaskState.LOST) { // 从taskIdToExecutorId中获取Task对应的
              Executor的身份标识
            val execId = taskIdToExecutorId.getOrElse(tid, throw new IllegalState
                Exception(
              "taskIdToTaskSetManager.contains(tid) <=> taskIdToExecutorId.
                  contains(tid)"))
            if (executorIdToRunningTaskIds.contains(execId)) {
              reason = Some(
                SlaveLost(s"Task $tid was lost, so marking the executor as lost
                    as well."))
              removeExecutor(execId, reason.get) // 移除Executor，移除的原因是
                  SlaveLost
              failedExecutor = Some(execId)
            }
          }
          if (TaskState.isFinished(state)) {
            cleanupTaskState(tid) // 清除Task在taskIdToTaskSetManager、taskIdTo
                ExecutorId中的数据
            taskSet.removeRunningTask(tid) // 减少正在运行的任务数量
            if (state == TaskState.FINISHED) { // 对执行成功的任务的结果进行处理
              taskResultGetter.enqueueSuccessfulTask(taskSet, tid, serializedData)
            } else if (Set(TaskState.FAILED, TaskState.KILLED, TaskState.LOST).
                contains(state)) {
              // 对执行失败的Task的结果进行处理
```

仅供非商业用途或交流学习使用


```
                taskResultGetter.enqueueFailedTask(taskSet, tid, state, serializedData)
              }
            }
          case None =>
            logError(
              ("Ignoring update with state %s for TID %s because its task set is gone
                  (this is " +
                "likely the result of receiving duplicate task finished status updates)
                    or its " +
                "executor has been marked as failed.")
                .format(state, tid))
        }
      } catch {
        case e: Exception => logError("Exception in statusUpdate", e)
      }
    }
    // 妥善安置丢失的Executor上正在运行的Task
    if (failedExecutor.isDefined) {
      assert(reason.isDefined)
      dagScheduler.executorLost(failedExecutor.get, reason.get)
      backend.reviveOffers()
    }
  }
```

根据代码清单7-103，statusUpdate方法的执行步骤如下。

1）从taskIdToTaskSetManager中获取Task对应的TaskSetManager。

2）如果要更新的任务状态是LOST，那么从taskIdToExecutorId中获取Task对应的Executor的身份标识。如果此Executor上正在运行Task，那么调用removeExecutor方法移除Executor，移除的原因是SlaveLost。

3）如果要更新的任务状态是完成状态（包括FINISHED、FAILED、KILLED、LOST），那么首先调用cleanupTaskState方法（见代码清单7-104），清除此Task在taskIdToTaskSetManager、taskIdToExecutorId等缓存中的数据，然后调用TaskSetManager的removeRunningTask方法（见代码清单7-71），减少正在运行的任务数量。如果任务状态是FINISHED，则调用TaskResultGetter的enqueueSuccessfulTask方法（见代码清单7-94），对执行成功的任务的结果进行处理。如果任务状态是FAILED、KILLED、LOST三者中的一种，那么调用TaskResultGetter的enqueueFailedTask方法（见代码清单7-95），对执行失败的Task的结果进行处理。

4）如果failedExecutor设置了Executor的身份标识，说明此Executor已经被移除，那么此Executor上正在运行的Task需要得到妥善的安置。安置的办法是：首先调用DagScheduler的executorLost方法（此方法非常简单，只是向DAGSchedulerEvent ProcessLoop投递了ExecutorLost消息，根据代码清单7-22的内容，DAGSchedulerEvent ProcessLoop处理ExecutorLost消息时，将调用DagScheduler的handleExecutorLost方法对丢失的Executor作进一步处理，限于篇幅，handleExecutorLost方法的实现留给感兴趣的读者自行研究），然后调用SchedulerBackend的reviveOffers方法给Task分配资源并运行Task。



仅供非商业用途或交流学习使用

代码清单7-104 清除Task的状态

```
private def cleanupTaskState(tid: Long): Unit = {
  taskIdToTaskSetManager.remove(tid)
  taskIdToExecutorId.remove(tid).foreach { executorId =>
    executorIdToRunningTaskIds.get(executorId).foreach { _.remove(tid) }
  }
}
```

上面专门分析了 TaskSchedulerImpl 的 statusUpdate 方法的实现。现在我们对 statusUpdate 方法的第 3）步进行聚焦，即专门来分析任务执行成功后的结果处理。根据 7.9.1 节对 TaskResultGetter 的 enqueueSuccessfulTask 方法的介绍，我们知道在获取到 Task 的执行结果后，将调用 TaskSchedulerImpl 的 handleSuccessfulTask 方法。TaskSchedulerImpl 的 handleSuccessfulTask 方法的实现如代码清单 7-105 所示。

代码清单7-105 TaskSchedulerImpl的handleSuccessfulTask方法

```
def handleSuccessfulTask(
    taskSetManager: TaskSetManager,
    tid: Long,
    taskResult: DirectTaskResult[_]): Unit = synchronized {
  taskSetManager.handleSuccessfulTask(tid, taskResult)
}
```

根据代码清单 7-105，handleSuccessfulTask 方法实际调用了 TaskSetManager 的 handleSuccessfulTask 方法（见代码清单 7-77），这样就把 Task 执行结果的处理整个串联了起来，如图 7-16 所示。

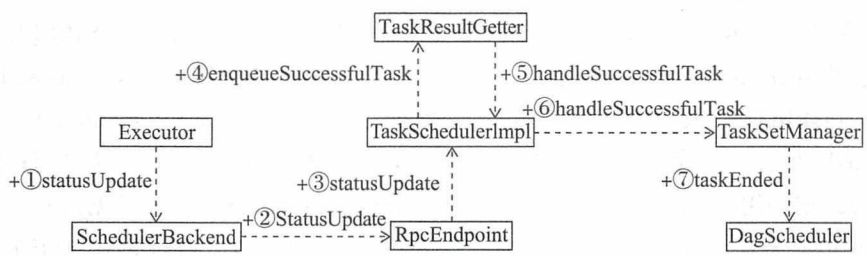


图 7-16 Task 执行结果的处理流程

这里对图 7-16 中的记号进行说明。

记号①：Executor 中的 TaskRunner 在执行 Task 的过程中，不断将 Task 的状态通过调用 SchedulerBackend 的实现类（比如 Local 模式下的 LocalSchedulerBackend 或 Standalone 模式下的 StandaloneSchedulerBackend）的 statusUpdate 方法告诉 SchedulerBackend 的实现类。当 Task 执行成功后，TaskRunner 也会将 Task 的完成状态告诉 SchedulerBackend 的实现类。

记号②：代表 SchedulerBackend 的实现类将 Task 的完成状态封装为 StatusUpdate 消息发送给 RpcEndpoint 的实现类（比如 Local 模式下的 LocalEndpoint 或 Standalone 模式下的

仅供非商业用途或交流学习使用

DriverEndpoint)。

记号③：RpcEndpoint 的实现类接收到 StatusUpdate 消息后，将调用 TaskSchedulerImpl 的 statusUpdate 方法。

记号④：TaskSchedulerImpl 的 statusUpdate 方法发现 Task 是执行成功的状态，那么调用 TaskResultGetter 的 enqueueSuccessfulTask 方法。

记号⑤：TaskResultGetter 的 enqueueSuccessfulTask 方法对 DirectTaskResult 类型的结果进行反序列化得到 Task 执行结果，对于 IndirectTaskResult 类型的结果需要先从远端下载 Block 数据，然后再进行反序列化得到 Task 执行结果。TaskResultGetter 获取到 Task 执行结果后，调用 TaskSchedulerImpl 的 handleSuccessfulTask 方法交给 TaskSchedulerImpl 处理。

记号⑥：TaskSchedulerImpl 的 handleSuccessfulTask 方法将直接调用 TaskSetManager 的 handleSuccessfulTask 方法。

记号⑦：TaskSetManager 的 handleSuccessfulTask 方法最重要的一步是调用 DAGScheduler 的 taskEnded 方法。对于 ResultTask 的结果，DAGScheduler 的 taskEnded 方法会将它交给 JobWaiter 的 resultHandler 函数来处理。对于 ShuffleMapTask 的结果，DAGScheduler 的 taskEnded 方法则将 Task 的 partitionId 和 MapStatus 追加到 Stage 的 outputLocs 中。如果没有待计算的分区，则需要将 Stage 的 shuffleId 和 outputLocs 中的 MapStatus 注册到 MapOutputTrackerMaster 的 mapStatuses 中。如果有某些分区的 Task 执行失败，则重新提交 ShuffleMapStage，否则调用 submitWaitingChildStages 方法提交当前 ShuffleMapStage 的子 Stage。



7.10.8 TaskSchedulerImpl 的常用方法

本节还将介绍一些 TaskSchedulerImpl 提供的常用方法。

1. removeExecutor

removeExecutor 方法（见代码清单 7-106）用于移除 Executor 及其对应的 Host 和机架等缓存信息。

代码清单7-106 移除Executor

```
private def removeExecutor(executorId: String, reason: ExecutorLossReason) {
  executorIdToRunningTaskIds.remove(executorId).foreach { taskIds =>
    logDebug("Cleaning up TaskScheduler state for tasks " +
      s"${taskIds.mkString("[", ",", "]")} on failed executor $executorId")
    taskIds.foreach(cleanupTaskState)
  }

  val host = executorIdToHost(executorId)
  val execs = hostToExecutors.getOrElse(host, new HashSet)
  execs -= executorId
  if (execs.isEmpty) {
    hostToExecutors -= host
    for (rack <- getRackForHost(host); hosts <- hostsByRack.get(rack)) {
      hosts -= host
```

仅供非商业用途或交流学习使用


```
        if (hosts.isEmpty) {
          hostsByRack -= rack
        }
      }
    }

    if (reason != LossReasonPending) {
      executorIdToHost -= executorId
      rootPool.executorLost(executorId, host, reason)
    }
  }
```

根据代码清单 7-106，removeExecutor 方法接收两个参数：Executor 的身份标识和移除 Executor 的原因（即 ExecutorLossReason）。ExecutorLossReason 有四个子类，分别代表四种不同的原因。

- SlaveLost：Worker 丢失。
- LossReasonPending：未知的原因导致的 Executor 退出。
- ExecutorKilled：Executor 被“杀死”了。
- ExecutorExited：Executor 退出了。

有了对 ExecutorLossReason 的了解，现在来看看 removeExecutor 方法的执行步骤。

1）从 executorIdToRunningTaskIds 移除 Executor 的缓存，并调用 cleanupTaskState 方法（见代码清单 7-104），清除在此 Executor 上正在运行的 Task 在 taskIdToTaskSetManager、taskIdToExecutorId 等缓存中的数据。



2）从 hostToExecutors 中移除此 Executor 的信息。

3）如果 hostToExecutors 中此 Executor 所在的 Host 主机已经没有任何 Executor 了，那么从 hostsByRack 中移除此 Host 的信息。

4）如果 hostsByRack 中此 Executor 所在的机架已经没有任何 Host 了，那么从 hostsByRack 中移除此机架的信息。

5）如果移除 Executor 的原因不是 LossReasonPending，那么首先从 executorIdToHost 中移除此 Executor 的缓存，然后调用根调度池 rootPool 的 executorLost 方法（见代码清单 7-51），将在此 Executor 上正在运行的 Task 作为失败任务处理，最后重新提交这些任务。

2. executorLost

executorLost 方法（见代码清单 7-107）用于处理 Executor 丢失。

代码清单7-107　处理Executor丢失

```
override def executorLost(executorId: String, reason: ExecutorLossReason): Unit = {
  var failedExecutor: Option[String] = None

  synchronized {
    if (executorIdToRunningTaskIds.contains(executorId)) {
      val hostPort = executorIdToHost(executorId)
      logExecutorLoss(executorId, hostPort, reason)
```

仅供非商业用途或交流学习使用


```
        removeExecutor(executorId, reason) // 移除Executor
        failedExecutor = Some(executorId) // 将failedExecutor设置为此Executor
      } else {
        executorIdToHost.get(executorId) match {
          case Some(hostPort) =>
            logExecutorLoss(executorId, hostPort, reason)
            removeExecutor(executorId, reason) // 移除Executor

          case None =>
            logError(s"Lost an executor $executorId (already removed): $reason")
        }
      }
    }
    if (failedExecutor.isDefined) {
      dagScheduler.executorLost(failedExecutor.get, reason)
      backend.reviveOffers() // 给Task分配资源并运行Task
    }
  }
```

根据代码清单7-107，executorLost方法的执行步骤如下。

1）如果executorIdToRunningTaskIds中包含指定的Executor的身份标识，这说明此时在此Executor上已经有Task正在运行，那么调用removeExecutor方法移除Executor，并将failedExecutor设置为此Executor。

2）如果executorIdToRunningTaskIds中不包含指定的Executor的身份标识，这说明此时在此Executor上没有Task正在运行，那么从executorIdToHost中获取此Executor对应的Host，并调用removeExecutor方法移除Executor。



3）如果failedExecutor设置了Executor的身份标识，这说明此Executor已经被移除，那么此Executor上正在运行的Task需要得到妥善的安置。安置的办法是：首先调用DagScheduler的executorLost方法（此方法非常简单，只是向DAGSchedulerEvent ProcessLoop投递了ExecutorLost消息，根据代码清单7-22的内容，DAGSchedulerEvent ProcessLoop处理ExecutorLost消息时，将调用DagScheduler的handleExecutorLost方法对丢失的Executor作进一步处理，限于篇幅，handleExecutorLost方法的实现留给感兴趣的读者自行研究），然后调用SchedulerBackend的reviveOffers方法给Task分配资源并运行Task。

3. postStartHook

由于TaskSchedulerImpl对任务资源的运行依赖于SchedulerBackend，所以为了避免TaskSchedulerImpl在SchedulerBackend准备就绪之前，就将Task交给SchedulerBackend处理，因此实现了postStartHook方法用于等待SchedulerBackend准备就绪，其实现如代码清单7-108所示。

代码清单7-108 postStartHook方法的实现

```
override def postStartHook() {
  waitBackendReady()
}
```

仅供非商业用途或交流学习使用


```
  private def waitBackendReady(): Unit = {
    if (backend.isReady) {
      return
    }
    while (!backend.isReady) {
      if (sc.stopped.get) {
        throw new IllegalStateException("Spark context stopped while waiting for backend")
      }
      synchronized {
        this.wait(100)
      }
    }
  }
```

7.11 小结

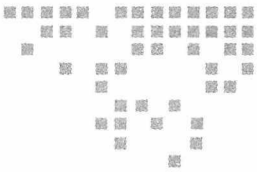
由于 RDD 是 Spark 调度系统的原材料，因此本章首先对 RDD 与 DAG 调度有关的功能进行了详细介绍。

通过对 DAGScheduler 处理 RDD 构成的 DAG、将 DAG 中的 RDD 划分为多个 Stage、提交 ResultStage、提交未计算的 Task 等功能的分析，DAGScheduler 的整个工作流程浮现在读者面前。OutputCommitCoordinator 是 DAGScheduler 中的重要组件，其实现方式简洁明了，有很高的借鉴意义。

TaskSchedulerImpl 依赖于 LauncherBackend 和 SchedulerBackend。通过对 TaskSchedulerImpl 的初始化、启动、提交 Task、资源分配等功能的分析，读者可以更加深入地了解 TaskSchedulerImpl 的调度流程。此外，通过理解调度算法、调度池、Task 本地性、推断执行等内容，将对 Spark 性能优化、二次开发带来收益。



仅供非商业用途或交流学习使用



第 8 章 Chapter 8

计算引擎

“一百馒头一百僧，大僧三个更无争，小僧三人分一个，大小和尚得几丁？”

——《算法统宗》



本章导读

《算法统宗》是明代珠算家程大位的著作，上面题目的意思是一百个馒头可以供一百个僧人食用，大僧人能吃三个，三个小僧人分着吃一个，那么大和尚和小和尚分别有多少人？按照一元一次方程式可以算得大僧人共二十五人，小僧人共七十五人。

任何一个数字原本是固定的，你若不去“动”它，它便不会有变化。这里的“动”可以认为是你列出的数学方程式，加上你大脑的思考，最后得出结果的过程。数学方程式是算法或工具，大脑思考是动力。Spark 的计算也离不开算法与动力，算法就是你使用 Spark API 开发出来的诸如 word count 一类的应用，而动力就是本章将要介绍的计算引擎。

Spark 任务的运行离不开 CPU 和内存，应用程序可以指定需要的资源总量。Spark 将计算内存分为堆外内存和堆内存，并且将这些内存分配给不同的 Task。每个 Task 中有很多组件作为内存消费者共享分配给 Task 的内存，所以 Spark 还将对 Task 的内存进行更细粒度的管理。

在所有 MapReduce 框架中，Shuffle 是连接 map 任务和 reduce 任务的桥梁。map 任务的中间输出要作为 reduce 任务的输入，就必须经过 Shuffle，所以 Shuffle 的性能优劣直接决定了整个计算引擎的性能和吞吐量。相比于 Hadoop 的 MapReduce，我们将看到 Spark 提供了多种计算结果处理的方式及对 Shuffle 过程进行的多种优化。

仅供非商业用途或交流学习使用

本章主要讲解的内容如下。

- 计算引擎概述。
- 执行内存。
- Tungsten。
- 任务内存管理器。
- Task 详解。
- IndexShuffleBlockResolver 详解。
- 特质 WritablePartitionedPairCollection。
- 采样与估算。
- 聚合缓存 AppendOnlyMap。
- 简单缓存 PartitionedPairBuffer。
- 外部排序器 ExternalSorter。
- Shuffle 管理器。

8.1 计算引擎概述

Spark 的计算引擎主要包括执行内存和 Shuffle 两部分。



1. 执行内存

执行内存主要包括执行内存、任务内存管理器（TaskMemoryManager）、内存消费者（MemoryConsumer）等内容。执行内存包括在 JVM 堆上进行分配的执行内存池（ExecutionMemoryPool）和在操作系统的内存中进行分配的 Tungsten。内存管理器将提供 API 对执行内存和 Tungsten 进行管理（包括申请内存、释放内存等）。因为同一节点上能够运行多次任务尝试，所以需要每一次任务尝试都有单独的任务内存管理器为其服务。任务尝试通过任务内存管理器与内存管理器交互，以申请任务尝试所需要的执行内存，并在任务尝试结束后释放使用的执行内存。一次任务尝试过程中会有多个组件需要使用执行内存，这些组件统称为内存消费者。内存消费者多种多样，有对 map 任务的中间输出数据在 JVM 堆上进行缓存、聚合、溢出、持久化等处理的 ExternalSorter，也有在操作系统内存中进行缓存、溢出、持久化处理的 ShuffleExternalSorter，还有将 key/value 对存储到连续的内存块中的 RowBasedKeyValueBatch。消费者需要的执行内存都是向任务内存管理器所申请的。从执行内存的角度来看，计算引擎的整体架构如图 8-1 所示。

2. 什么是 Shuffle

Shuffle 是所有 MapReduce 计算框架必须面临的执行阶段，Shuffle 用于打通 map 任务的输出与 reduce 任务的输入，map 任务的中间输出结果按照指定的分区策略（例如，按照 key 值哈希）分配给处理某一个分区的 reduce 任务，这个过程如图 8-2 所示。

仅供非商业用途或交流学习使用

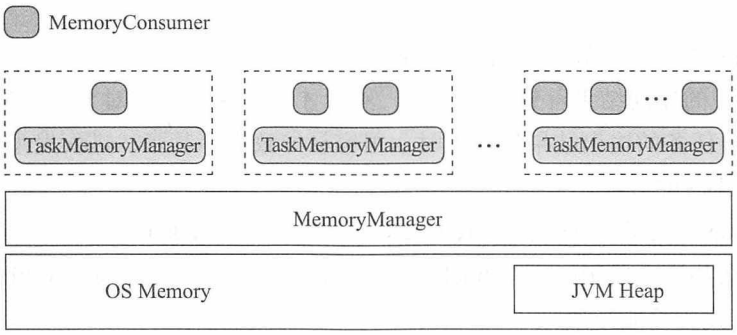


图 8-1 计算引擎的执行内存体系

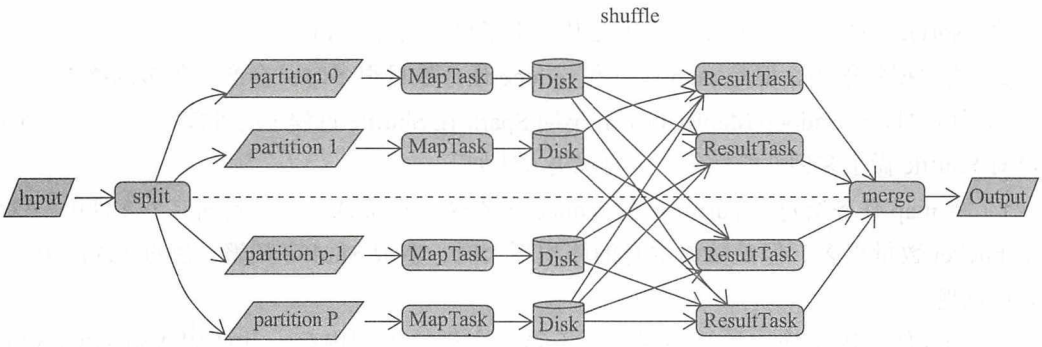


图 8-2 MapReduce 计算框架的 Shuffle 过程

在具体分析源码之前，我们先看看 Spark 早期版本的 Shuffle 是怎样的，如图 8-3 所示。

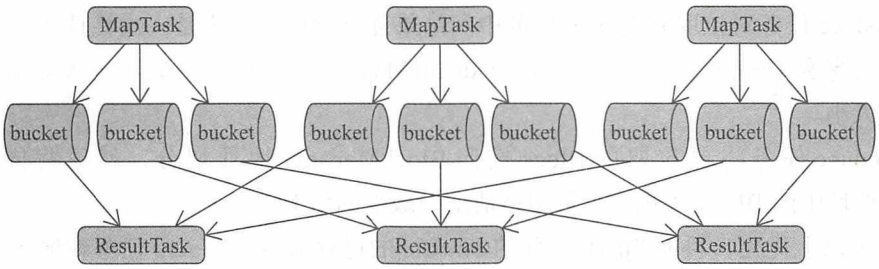


图 8-3 Spark 早期版本的 Shuffle 过程

这里对图 8-3 做一些解释。

1）map 任务会为每一个 reduce 任务创建一个 bucket。假设有 M 个 map 任务，R 个 reduce 任务，则 map 阶段一共会创建 M×R 个桶（bucket）。

2）map 任务会将产生的中间结果按照分区（partition）写入到不同的 bucket 中。

3）reduce 任务从本地或者远端的 map 任务所在的 BlockManager 获取相应的 bucket 作

仅供非商业用途或交流学习使用

为输入。

Spark 早期版本的 Shuffle 过程存在以下问题。

1）map 任务的中间结果首先存入内存，然后才写入磁盘。这对于内存的开销很大，当一个节点上 map 任务的输出结果集很大时，很容易导致内存紧张，进而发生内存溢出（Out Of Memory，简称 OOM）。

2）每个 map 任务都会输出 R（reduce 任务数量）个 bucket。假如 M 等于 1000，R 也等于 1000，那么共计生成 100 万个 bucket，在 bucket 本身不大，但是 Shuffle 很频繁的情况下，磁盘 I/O 将成为性能瓶颈。

熟悉 Hadoop 的读者应该知道，Hadoop MapReduce 的 Shuffle 过程存在以下问题。

1）reduce 任务获取到 map 任务的中间输出后，会对这些数据在磁盘上进行合并（merge）和排序（sort），虽然占用内存很小，但是却产生了更多的磁盘 I/O。

2）当数据量很小，但是 map 任务和 reduce 任务数目很多时，会产生很多网络 I/O。

为了解决以上 Hadoop MapReduce 和早期 Spark 在 Shuffle 过程中的性能问题，目前 Spark 已经对 Shuffle 做了多种性能优化，主要解决方法如下。

1）将 map 任务给每个 partition 的 reduce 任务输出的 bucket 合并到同一个文件中，这解决了 bucket 数量很多，但是数据本身的体积不大时，造成 Shuffle 频繁，磁盘 I/O 成为性能瓶颈的问题。



2）map 任务逐条输出计算结果，而不是一次性输出到内存，并使用 AppendOnlyMap 缓存及其聚合算法对中间结果进行聚合，这大大减小了中间结果所占的内存大小。

3）对 SizeTrackingAppendOnlyMap、SizeTrackingPairBuffer 及 Tungsten 的 Page 进行溢出判断，当超出溢出限制的大小时，将数据写入磁盘，防止内存溢出。

4）reduce 任务对拉取到的 map 任务中间结果逐条读取，而不是一次性读入内存，并在内存中进行聚合（其本质上也使用了 AppendOnlyMap 缓存）和排序，这也大大减小了数据占用的内存大小。

5）reduce 任务将要拉取的 Block 按照 BlockManager 地址划分，然后将同一 BlockManager 地址中的 Block 累积为少量网络请求，减少网络 I/O。

经过了以上优化，目前 Spark 实现的 Shuffle 的过程大致为：map 任务在输出时会进行分区计算并生成数据文件和索引文件等步骤，可能还伴随有缓存、排序、聚合、溢出、合并等操作。reduce 任务将 map 任务输出的 Block 划分为本地和远端的 Block，对于远端的 Block，需要使用 ShuffleClient 从远端节点下载，而对于本地的 Block，只需要从本地的存储体系中读取即可。reduce 任务读取到 map 任务输出的数据后，可能进行缓存、排序、聚合、溢出、合并等操作，最终输出结果。

在接下来的源码分析过程中，我们一起来看看这些解决方法是如何实现的。

8.2 内存管理器与执行内存

在第 6 章介绍 MemoryManager 时，只介绍了存储内存的内容，而执行内存的内容由于属于计算引擎的一部分，所以放到本节来介绍。

8.2.1 ExecutionMemoryPool 详解

ExecutionMemoryPool 继承了 MemoryPool，是执行内存池的具体实现。ExecutionMemoryPool 只是逻辑上的执行内存池，并不是堆上和堆外的实际内存。要理解 ExecutionMemoryPool，应该从其属性开始。ExecutionMemoryPool 继承了 MemoryPool 的 lock 和 _poolSize 两个属性，还增加了一些特有的属性。

- memoryMode：内存模式（MemoryMode）。根据此属性可以断定用于执行的内存池包括堆内存和堆外内存两种。
- poolName：内存池的名称。如果 memoryMode 是 MemoryMode.ON_HEAP，则内存池名称为 on-heap execution。如果 memoryMode 是 MemoryMode.OFF_HEAP，则内存池名称为 off-heap execution。
- memoryForTask：任务尝试的身份标识（taskAttemptId）与所消费内存的大小之间的映射关系。

了解了 ExecutionMemoryPool 的属性，现在来看看 ExecutionMemoryPool 中的方法。

- memoryUsed：已经使用的内存大小（单位为字节）。实际为所有任务尝试所消费的内存大小之和，即 memoryForTask 这个 Map 中所有 value 的和。
- getMemoryUsageForTask：获取任务尝试使用的内存大小，即 memoryForTask 中 taskAttemptId 对应的 value 值。

以上方法的实现都很简单，下面将介绍一些需要详细分析的方法。

1. acquireMemory

acquireMemory 方法（见代码清单 8-1）用于给 taskAttemptId 对应的任务尝试获取指定大小（即 numBytes）的内存。

代码清单8-1 获取任务尝试所需的内存

```
private[memory] def acquireMemory(
    numBytes: Long,
    taskAttemptId: Long,
    maybeGrowPool: Long => Unit = (additionalSpaceNeeded: Long) => Unit,
    computeMaxPoolSize: () => Long = () => poolSize): Long = lock.synchronized {
  assert(numBytes > 0, s"invalid number of bytes requested: $numBytes")

  if (!memoryForTask.contains(taskAttemptId)) {
    memoryForTask(taskAttemptId) = 0L // 将taskAttemptId放入memoryForTask
    lock.notifyAll()
  }
```

```
    while (true) {
      val numActiveTasks = memoryForTask.keys.size // 获取当前激活的Task的数量
      val curMem = memoryForTask(taskAttemptId) //获取当前任务尝试所消费的内存
      maybeGrowPool(numBytes - memoryFree) // 从StorageMemoryPool回收或借用内存
      val maxPoolSize = computeMaxPoolSize() // 计算内存池的最大大小
      val maxMemoryPerTask = maxPoolSize / numActiveTasks // 计算每个任务尝试最大可以使
                                                          // 用的内存大小
      val minMemoryPerTask = poolSize / (2 * numActiveTasks) // 计算每个任务尝试最小保
                                                             // 证使用的内存大小
      val maxToGrant = math.min(numBytes, math.max(0, maxMemoryPerTask - curMem))
      val toGrant = math.min(maxToGrant, memoryFree) // 计算当前任务尝试真正可以申请获取
                                                     // 的内存大小

      if (toGrant < numBytes && curMem + toGrant < minMemoryPerTask) {
        logInfo(s"TID $taskAttemptId waiting for at least 1/2N of $poolName pool to be
            free")
        lock.wait() // 内存不足，使当前线程处于等待状态
      } else {
        memoryForTask(taskAttemptId) += toGrant // 成功获取到toGrant指定大小的内存
        return toGrant
      }
    }
    0L  // Never reached
  }
```

根据代码清单 8-1，acquireMemory 方法的执行步骤如下。

1）如果 memoryForTask 中还没有记录 taskAttemptId，则将 taskAttemptId 放入 memory ForTask，并且 taskAttemptId 所消费的内存为 0。唤醒其他等待获取 Execution MemoryPool 的锁的线程。

2）不断循环执行以下操作。

① 获取当前激活的 Task 的数量（即 numActiveTasks）。

② 获取当前任务尝试所消费的内存（即 curMem）。

③ 调用函数 maybeGrowPool（此函数参数的具体实现可以参考代码清单 8-8），以回收 StorageMemoryPool 从当前 ExecutionMemoryPool 借用的内存，而且在 StorageMemoryPool 还有空闲空间时，从 StorageMemoryPool 借用内存，这可能导致 StorageMemoryPool 中的一些 Block 被驱逐出去。

④ 调用函数 computeMaxPoolSize（此函数参数的具体实现可以参考代码清单 8-8），计算内存池的最大大小（即 maxPoolSize）。

⑤ 计算每个任务尝试最大可以使用的内存大小（即 maxMemoryPerTask）。

⑥ 计算每个任务尝试最小保证使用的内存大小（即 minMemoryPerTask）。

⑦ 计算当前任务尝试真正可以申请获取的内存大小（即 toGrant）。

⑧ 如果 toGrant 小于任务尝试本来要申请的内存大小（即 numBytes），并且 curMem 与 toGrant 的和小于 minMemoryPerTask，则使得当前线程处于等待状态。这说明如果任务尝试要申请的内存得不到满足，甚至连每个任务尝试最基本的内存大小都得不到满足，则需

要等待其他任务尝试释放内存。当其他任务尝试释放内存后，会进入下一次循环，直到获取到满意的内存大小。

⑨ 如果 toGrant 大于等于 numBytes（即真正可以申请获取的内存超出了期望获得的内存时）或者 toGrant 与 curMem 的和大于等于 minMemoryPerTask（即任务尝试得到了最基本的内存保证），那么在 memoryForTask 中增加 taskAttemptId 所代表任务尝试的消费内存大小并返回 toGrant 退出循环。

小贴士： 以上对 acquireMemory 方法的步骤拆解，读者可能并不容易理解。这里对它的处理逻辑作个简单总结：假设当前有 N 个线程，必须保证每个线程在溢出之前至少获得 1/2N 的内存，并且每个线程最多获得 1/N 的内存。由于 N 是动态变化的变量，所以要持续对这些线程进行跟踪，以便于线程的数量发生变化时，重新按照 1/2N 和 1/N 计算。

2. releaseMemory

releaseMemory 方法（见代码清单 8-2）用于给 taskAttemptId 对应的任务尝试释放指定大小（即 numBytes）的内存。

代码清单8-2 释放任务尝试指定大小的内存

```
def releaseMemory(numBytes: Long, taskAttemptId: Long): Unit = lock.synchronized {
  val curMem = memoryForTask.getOrElse(taskAttemptId, 0L) // 获取任务尝试所消费的内存
  var memoryToFree = if (curMem < numBytes) { // 计算能够释放的内存大小
    logWarning(
      s"Internal error: release called on $numBytes bytes but task only has $curMem
          bytes " +
        s"of memory from the $poolName pool")
    curMem
  } else {
    numBytes
  }
  if (memoryForTask.contains(taskAttemptId)) { // 释放内存（逻辑内存）
    memoryForTask(taskAttemptId) -= memoryToFree
    if (memoryForTask(taskAttemptId) <= 0) {
      memoryForTask.remove(taskAttemptId)
    }
  }
  lock.notifyAll() // 唤醒所有申请获得内存，但是处于等待状态的线程
}
```

根据代码清单 8-2，releaseMemory 方法的执行步骤如下。

1）获取 taskAttemptId 代表的任务尝试所消费的内存（即 curMem）。

2）如果 curMem 小于 numBytes，则真正要释放的内存大小（即 memoryToFree）等于 curMem，否则 memoryToFree 等于 numBytes。

3）taskAttemptId 代表的任务尝试占用的内存大小减去 memoryToFree。如果 taskAttemptId 代表的任务尝试占用的内存大小小于等于零，还需要将 taskAttemptId 与所消费内存的映射关系从 memoryForTask 中清除。

4）唤醒所有因为调用 acquireMemory 方法申请获得内存，却因为内存不足处于等待状态的线程。

3. releaseAllMemoryForTask

releaseAllMemoryForTask 方法（见代码清单 8-3）用于释放 taskAttemptId 对应的任务尝试所消费的所有内存。

代码清单8-3 释放任务尝试的所有内存

```
def releaseAllMemoryForTask(taskAttemptId: Long): Long = lock.synchronized {
  val numBytesToFree = getMemoryUsageForTask(taskAttemptId)
  releaseMemory(numBytesToFree, taskAttemptId)
  numBytesToFree
}
```

根据代码清单 8-3，releaseAllMemoryForTask 方法首先调用 getMemoryUsageForTask 方法获取 taskAttemptId 对应的任务尝试消费的内存，然后调用 releaseMemory 方法释放，最后返回释放的内存大小。

8.2.2 MemoryManager 模型与执行内存

在 6.5.3 节曾经详细介绍了 MemoryManager 的内存模型，但是没有介绍其与执行内存相关的方法。本节将介绍 MemoryManager 中与执行相关的方法。

1）acquireExecutionMemory：为执行 taskAttemptId 对应的任务尝试，从堆内存或堆外内存获取所需大小（即 numBytes）的内存。此方法的接口定义如下。

```
private[memory]  def acquireExecutionMemory(numBytes: Long, taskAttemptId: Long,
  memoryMode: MemoryMode): Long
```

2）releaseExecutionMemory：从堆内存或堆外内存释放 taskAttemptId 对应的任务尝试所消费的指定大小（即 numBytes）的执行内存。此方法的实现如代码清单 8-4 所示。

代码清单8-4 releaseExecutionMemory的实现

```
private[memory]
def releaseExecutionMemory(
    numBytes: Long,
    taskAttemptId: Long,
    memoryMode: MemoryMode): Unit = synchronized {
  memoryMode match {
    case MemoryMode.ON_HEAP => onHeapExecutionMemoryPool.releaseMemory(numBytes,
        taskAttemptId)
    case MemoryMode.OFF_HEAP => offHeapExecutionMemoryPool.releaseMemory(numBytes,
        taskAttemptId)
  }
}
```

3）releaseAllExecutionMemoryForTask：从堆内存及堆外内存释放 taskAttemptId 代表

的任务尝试所消费的所有执行内存。此方法的实现如代码清单 8-5 所示。

代码清单8-5 releaseAllExecutionMemoryForTask的实现

```
private[memory] def releaseAllExecutionMemoryForTask(taskAttemptId: Long): Long =
    synchronized {
    onHeapExecutionMemoryPool.releaseAllMemoryForTask(taskAttemptId) +
    offHeapExecutionMemoryPool.releaseAllMemoryForTask(taskAttemptId)
}
```

4）executionMemoryUsed：获取堆上执行内存池与堆外执行内存池已经使用的执行内存之和。此方法的实现如代码清单 8-6 所示。

代码清单8-6 executionMemoryUsed的实现

```
final def executionMemoryUsed: Long = synchronized {
  onHeapExecutionMemoryPool.memoryUsed + offHeapExecutionMemoryPool.memoryUsed
}
```

5）getExecutionMemoryUsageForTask：获取 taskAttemptId 代表的任务尝试在堆上执行内存池与堆外执行内存池所消费的执行内存之和。此方法的实现如代码清单 8-7 所示。

代码清单8-7 getExecutionMemoryUsageForTask的实现

```
private[memory] def getExecutionMemoryUsageForTask(taskAttemptId: Long): Long =
    synchronized {
  onHeapExecutionMemoryPool.getMemoryUsageForTask(taskAttemptId) +
    offHeapExecutionMemoryPool.getMemoryUsageForTask(taskAttemptId)
}
```

8.2.3 UnifiedMemoryManager 与执行内存

在 6.5.4 节介绍 UnifiedMemoryManager 时，只介绍了与存储体系相关的方法，本节将介绍 UnifiedMemoryManager 提供的与执行内存相关的 acquireExecutionMemory 方法。

在 MemoryManager 中定义了 acquireExecutionMemory 方法的接口，需要子类去实现，UnifiedMemoryManager 对此方法的实现如代码清单 8-8 所示。

代码清单8-8 获取执行内存

```
override private[memory] def acquireExecutionMemory(
    numBytes: Long,
    taskAttemptId: Long,
    memoryMode: MemoryMode): Long = synchronized {
  assertInvariants()
  assert(numBytes >= 0)
  val (executionPool, storagePool, storageRegionSize, maxMemory) = memoryMode match {
    case MemoryMode.ON_HEAP => (
      onHeapExecutionMemoryPool,
      onHeapStorageMemoryPool,
      onHeapStorageRegionSize,
      maxHeapMemory)
```

```
      case MemoryMode.OFF_HEAP => (
        offHeapExecutionMemoryPool,
        offHeapStorageMemoryPool,
        offHeapStorageMemory,
        maxOffHeapMemory)
    }

    def maybeGrowExecutionPool(extraMemoryNeeded: Long): Unit = { // 此函数用于借用或收
        回存储内存
      if (extraMemoryNeeded > 0) {
        val memoryReclaimableFromStorage = math.max(
          storagePool.memoryFree,
          storagePool.poolSize - storageRegionSize)
        if (memoryReclaimableFromStorage > 0) {
          val spaceToReclaim = storagePool.freeSpaceToShrinkPool(
            math.min(extraMemoryNeeded, memoryReclaimableFromStorage))
          storagePool.decrementPoolSize(spaceToReclaim)
          executionPool.incrementPoolSize(spaceToReclaim)
        }
      }
    }

    def computeMaxExecutionPoolSize(): Long = {
      maxMemory - math.min(storagePool.memoryUsed, storageRegionSize)
    }

    executionPool.acquireMemory( // 任务尝试获取指定大小（即numBytes）的内存
      numBytes, taskAttemptId, maybeGrowExecutionPool, computeMaxExecutionPoolSize)
  }
```

根据代码清单 8-8，UnifiedMemoryManager 的 acquireExecutionMemory 方法的执行步骤如下。

1）根据内存模式获取 UnifiedMemoryManager 中管理的堆上或堆外的执行内存池（executionPool）、存储内存池（storagePool）、存储区域大小（storageRegionSize）、内存最大值（maxMemory）。

2）调用 ExecutionMemoryPool 的 acquireMemory 方法（见代码清单 8-1），给 taskAttemptId 对应的任务尝试获取指定大小（即 numBytes）的内存。这里以 maybeGrowExecutionPool 作为 acquireMemory 方法的函数参数 maybeGrowPool，以 computeMaxExecutionPoolSize 作为 acquireMemory 方法的函数参数 computeMaxPoolSize。

小贴士：根据 maybeGrowExecutionPool 的实现，如果存储内存池的空闲空间大于存储内存池从执行内存池借用的空间大小，那么除了回收被借用的空间外，还会向存储池再借用一些空间；如果存储池的空闲空间小于等于存储池从执行池借用的空间大小，那么只需要回收被借用的空间。根据 computeMaxExecutionPoolSize 方法的实现，我们知道计算最大的执行内存池时，如果存储区域的边界大小大于已经被存储使用的内存，那么执行内存的最大空间可以跨越存储内存与执行内存之间的“软”边界；如果存储区域的边界大小小于

等于已经被存储使用的内存，这说明存储内存已经跨越了存储内存与执行内存之间的“软”边界，执行内存可以收回被存储内存借用的空间。

8.3 内存管理器与 Tungsten

什么是 Tungsten？翻译为中文是“钨”的意思。Tungsten 最早是由 Databricks 公司提出的对 Spark 的内存和 CPU 使用进行优化的计划，但本书限定 Tungsten 是一种内存分配与释放的实现。Tungsten 使用 sun.misc.Unsafe 的 API 直接操作系统内存，避免了在 JVM 中加载额外的 Class，也不用创建额外的对象，因而减少了不必要的内存开销，降低了 GC 扫描和回收的频率，提升了处理性能。堆外内存可以被精确地申请和释放，而且序列化的数据占用的空间可以被精确计算，所以相比堆内存来说降低了管理的难度，也降低了误差。

8.3.1 MemoryBlock 详解

操作系统中的 Page 是一个内存块，在 Page 中可以存放数据，操作系统中会有多种不同的 Page。操作系统对数据的读取，往往是先确定数据所在的 Page，然后使用 Page 的偏移量（offset）和所读取数据的长度（length）从 Page 中读取数据。

在 Tungsten 中实现了一种与操作系统的内存 Page 非常相似的数据结构，这个对象就是 MemoryBlock。MemoryBlock 中的数据可能位于 JVM 的堆上，也可能位于 JVM 的堆外内存（操作系统内存）中。

由于 MemoryBlock 继承自 MemoryLocation，所以分析 MemoryBlock 之前，应该首先弄清楚 MemoryLocation。MemoryLocation 用于表示内存的位置信息。Tungsten 如果是堆外模式，那么 MemoryLocation 的实现如下。

```
public class MemoryLocation {
  @Nullable
  Object obj;
  long offset;
  public MemoryLocation(@Nullable Object obj, long offset) {
    this.obj = obj;
    this.offset = offset;
  }
  public MemoryLocation() {
    this(null, 0);
  }
  public void setObjAndOffset(Object newObj, long newOffset) {
    this.obj = newObj;
    this.offset = newOffset;
  }
  public final Object getBaseObject() {
    return obj;
```

```
  }
  public final long getBaseOffset() {
    return offset;
  }
}
```

可以看到，MemoryLocation 主要由 obj 和 offset 两个属性及其读写方法组成。有些读者可能会发现，obj 属性由注解 Nullable 来标注，这是为什么？ Tungsten 处于堆内存模式时，数据作为对象存储在 JVM 的堆上，此时的 obj 不为空。Tungsten 处于堆外内存模式时，数据存储在 JVM 的堆外内存（操作系统内存）中，因而不会在 JVM 中存在对象。offset 属性主要用来定位数据。当 Tungsten 处于堆内存模式时，首先从堆内找到对象，然后使用 offset 定位数据的具体位置。当 Tungsten 处于堆外内存模式时，则直接使用 offset 从堆外内存中定位。

定位到数据的位置后，该怎样读取数据呢？ MemoryBlock（见代码清单 8-9）继承自 MemoryLocation，代表从 obj 和 offset 定位的起始位置开始，固定长度（由 MemoryBlock 的 length 属性确定）的连续内存块。

代码清单8-9 MemoryBlock的实现

```
public class MemoryBlock extends MemoryLocation {
  private final long length;
  public int pageNumber = -1;
  public MemoryBlock(@Nullable Object obj, long offset, long length) {
    super(obj, offset);
    this.length = length;
  }
  public long size() {
    return length;
  }
  public static MemoryBlock fromLongArray(final long[] array) {
    return new MemoryBlock(array, Platform.LONG_ARRAY_OFFSET, array.length * 8L);
  }
  public void fill(byte value) {
    Platform.setMemory(obj, offset, length, value);
  }
}
```

根据代码清单 8-9，MemoryBlock 中一共有以下两个属性。

1）length：当前 MemoryBlock 的连续内存块的长度。

2）pageNumber：当前 MemoryBlock 的页号。TaskMemoryManager 分配由 MemoryBlock 表示的 Page 时，将使用此属性。

MemoryBlock 中提供了以下三个方法。

1）size：MemoryBlock 的大小，即 length。

2）fromLongArray：创建一个指向由长整型数组使用的内存的 MemoryBlock。

3）fill：以指定的字节填充整个 MemoryBlock，即将 obj 对象从 offset 开始，长度为

仅供非商业用途或交流学习使用

length 的堆内存替换为指定字节的值。Platform 中封装了对 sun.misc.Unsafe[⊖]的 API 调用，Platform 的 setMemory 方法实际调用了 sun.misc.Unsafe 的 setMemory 方法。

8.3.2 MemoryManager 模型与 Tungsten

MemoryManager 中除了存储内存和执行内存外，还定义了几个与 Tungsten 优化相关的常量。

1）tungstenMemoryMode：Tungsten 的内存模式。tungstenMemoryMode 也采用枚举类型 MemoryMode 来表示堆内存和堆外内存。当 Tungsten 在堆内存模式下，数据存储在 JVM 堆上，这时 Tungsten 选择 onHeapExecutionMemoryPool 作为内存池。当 Tungsten 在堆外内存模式下，数据则会存储在堆外内存（操作系统内存）中，这时 Tungsten 选择 offHeapExecution MemoryPool 作为内存池。可以通过 spark.memory.offHeap.enabled 属性（默认为 false）来配置是否启用 Tungsten 的堆外内存。

2）pageSizeBytes：Tungsten 采用的 Page 的默认大小（单位为字节）。可通过 spark.buffer.pageSize 属性进行配置。如果未指定 spark.buffer.pageSize 属性，则计算 pageSizeBytes 的方式如代码清单 8-10 所示。

代码清单8-10 pageSizeBytes的计算

```
val pageSizeBytes: Long = {
  val minPageSize = 1L * 1024 * 1024    // 1MB
  val maxPageSize = 64L * minPageSize  // 64MB
  val cores = if (numCores > 0) numCores else Runtime.getRuntime.available
      Processors()
  val safetyFactor = 16
  val maxTungstenMemory: Long = tungstenMemoryMode match {
    case MemoryMode.ON_HEAP => onHeapExecutionMemoryPool.poolSize
    case MemoryMode.OFF_HEAP => offHeapExecutionMemoryPool.poolSize
  }
  val size = ByteArrayMethods.nextPowerOf2(maxTungstenMemory / cores / safetyFactor)
  val default = math.min(maxPageSize, math.max(minPageSize, size))
  conf.getSizeAsBytes("spark.buffer.pageSize", default)
}
```



3）tungstenMemoryAllocator：Tungsten 采用的内存分配器（MemoryAllocator）。如果 tungstenMemoryMode 为 MemoryMode.ON_HEAP，那么 tungstenMemoryAllocator 为堆内存分配器（HeapMemoryAllocator），否则为使用 sun.misc.Unsafe 的 API 分配操作系统内存的分配器 UnsafeMemoryAllocator。

8.3.3 Tungsten 的内存分配器

MemoryAllocator 是 Tungsten 的内存分配器的接口规范，其定义如下。

⊖ sun.misc.Unsafe 不是 JDK 中的 API，而是 sun 公司内部使用的 API。sun.misc.Unsafe 中有很多 Native 的方法，用以直接操作内存。sun.misc.Unsafe 已经被声明为已弃用（Deprecated），但是很多开源项目依然使用它来提升性能。sun.misc.Unsafe 没有官方文档，但从网络上可以找到很多介绍它的文章。

仅供非商业用途或交流学习使用


```
public interface MemoryAllocator {
  MemoryBlock allocate(long size) throws OutOfMemoryError;
  void free(MemoryBlock memory);
  MemoryAllocator UNSAFE = new UnsafeMemoryAllocator();
  MemoryAllocator HEAP = new HeapMemoryAllocator();
}
```

根据上述代码，MemoryAllocator 定义了两个接口方法。

1）allocate：分配指定大小的连续内存块。按照这种方式分配的内存不能保证被清零，如果需要，可以在 MemoryBlock 上调用 fill(0)。

2）free：释放连续的内存块。

UnsafeMemoryAllocator 和 HeapMemoryAllocator 是在不同内存模式下，MemoryAllocator 的实现类。

1. HeapMemoryAllocator

HeapMemoryAllocator 是 Tungsten 在堆内存模式下使用的内存分配器，与 onHeapExecution MemoryPool 配合使用。HeapMemoryAllocator 只有一个属性。

```
@GuardedBy("this")
private final Map<Long, LinkedList<WeakReference<MemoryBlock>>> bufferPoolsBySize =
  new HashMap<>();
```

bufferPoolsBySize 是关于 MemoryBlock 的弱引用[⊖]的缓冲池，用于 Page 页（即 MemoryBlock）的分配。



HeapMemoryAllocator 实现了以下方法。

1）shouldPool: 用于判断对于指定大小的 MemoryBlock，是否需要采用池化机制（即从缓冲池 bufferPoolsBySize 中获取 MemoryBlock 或将 MemoryBlock 放入 bufferPoolsBySize）。根据 shouldPool 方法（见代码清单 8-11）的实现，当要分配的内存大小大于等于 1MB（常量 POOLING_THRESHOLD_BYTES 的值）时，需要从 bufferPoolsBySize 中获取 MemoryBlock。

代码清单8-11　shouldPool的实现

```
private boolean shouldPool(long size) {
  return size >= POOLING_THRESHOLD_BYTES;
}
```

2）allocate：用于分配指定大小（size）的 MemoryBlock。allocate 方法的实现如代码清单 8-12 所示。

代码清单8-12　HeapMemoryAllocator的allocate方法

```
@Override
public MemoryBlock allocate(long size) throws OutOfMemoryError {
```

⊖ 通过 WeakReference 来引用 MemoryBlock，当 MemoryBlock 不再被引用时，主动调用 System.gc() 可以保证被回收，以便加快堆内存的回收和分配效率。

仅供非商业用途或交流学习使用


```
  if (shouldPool(size)) {
    synchronized (this) {
      final LinkedList<WeakReference<MemoryBlock>> pool = bufferPoolsBySize.get(size);
      if (pool != null) {
        while (!pool.isEmpty()) {
          final WeakReference<MemoryBlock> blockReference = pool.pop();
          final MemoryBlock memory = blockReference.get();
          if (memory != null) {
            assert (memory.size() == size);
            return memory; // 从MemoryBlock的缓存中获取指定大小的MemoryBlock
          }
        }
        bufferPoolsBySize.remove(size); //没有指定大小的MemoryBlock，移除指定大小的
                                        //MemoryBlock缓存
      }
    }
  }
  long[] array = new long[(int) ((size + 7) / 8)];
  MemoryBlock memory = new MemoryBlock(array, Platform.LONG_ARRAY_OFFSET, size);
  if (MemoryAllocator.MEMORY_DEBUG_FILL_ENABLED) {
    memory.fill(MemoryAllocator.MEMORY_DEBUG_FILL_CLEAN_VALUE);
  }
  return memory; // 返回创建的MemoryBlock
}
```

根据代码清单 8-12，allocate 方法的执行步骤如下。

① 如果指定大小（size）的 MemoryBlock 需要采用池化机制，则从 bufferPoolsBySize 的弱引用中获取指定大小的 MemoryBlock。如果 bufferPoolsBySize 中没有指定大小的 MemoryBlock，则将指定大小的弱引用池从 bufferPoolsBySize 中移除。

② 如果指定大小（size）的 MemoryBlock 不需要采用池化机制或者 bufferPoolsBySize 中没有指定大小的 MemoryBlock，则创建 MemoryBlock 并返回。

3）free：用于释放 MemoryBlock。free 方法的实现如代码清单 8-13 所示。

代码清单8-13 HeapMemoryAllocator的free方法

```
@Override
public void free(MemoryBlock memory) {
  final long size = memory.size(); // 获取待释放MemoryBlock的大小
  if (MemoryAllocator.MEMORY_DEBUG_FILL_ENABLED) {
    memory.fill(MemoryAllocator.MEMORY_DEBUG_FILL_FREED_VALUE);
  }
  if (shouldPool(size)) {
    synchronized (this) {
      LinkedList<WeakReference<MemoryBlock>> pool = bufferPoolsBySize.get(size);
      if (pool == null) {
        pool = new LinkedList<>();
        bufferPoolsBySize.put(size, pool);
      }
      pool.add(new WeakReference<>(memory)); // 将MemoryBlock的弱引用放入buffer
          PoolsBySize中
    }
  } else {
```

仅供非商业用途或交流学习使用


```
        // Do nothing
      }
    }
```

根据代码清单 8-13，free 方法的执行步骤如下。

1）获取待释放 MemoryBlock 的大小。

2）如果 MemoryBlock 的大小需要采用池化机制，那么将 MemoryBlock 的弱引用放入 bufferPoolsBySize 中。

根据以上分析，HeapMemoryAllocator 对内存的分配的确是基于 JVM 的堆内存的，图 8-4 很好地表示了 HeapMemoryAllocator 的工作原理。

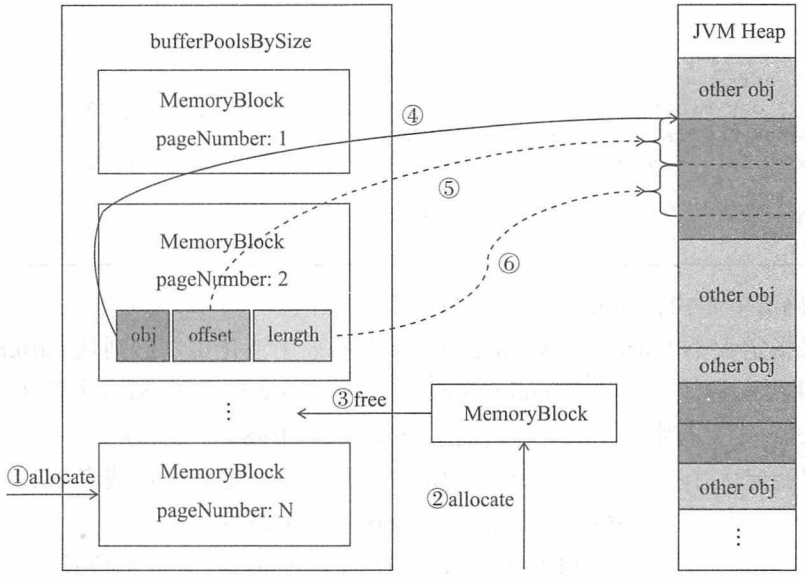


图 8-4 HeapMemoryAllocator 的工作原理

这里对图 8-4 进行一些介绍。

标记①：在分配 MemoryBlock 时，申请的大小大于等于 1MB（常量 POOLING_THRESHOLD_BYTES 的值），且 bufferPoolsBySize 中存在指定大小的 MemoryBlock，则从 bufferPoolsBySize 中获取 MemoryBlock。

标记②：在分配 MemoryBlock 时，申请的大小小于 1MB（常量 POOLING_THRESHOLD_BYTES 的值），或者 bufferPoolsBySize 中不存在指定大小的 MemoryBlock，则单独创建 MemoryBlock 用于分配。

标记③：在释放 MemoryBlock 时，如果 MemoryBlock 的大小大于等于 1MB（常量 POOLING_THRESHOLD_BYTES 的值），则将此 MemoryBlock 放入 bufferPoolsBySize 中。

标记④：MemoryBlock 的 obj 属性保存了对象在 JVM 堆中的地址。

标记⑤：MemoryBlock 的 offset 属性保存了 Page 的起始地址（即相对于所在对象在

仅供非商业用途或交流学习使用

JVM 堆中地址的偏移量）。

标记⑥：MemoryBlock 的 length 属性保存了 Page 的页面大小（即从 offset 开始，连续内存空间的大小）。

2. UnsafeMemoryAllocator

UnsafeMemoryAllocator 是 Tungsten 在堆外内存模式下使用的内存分配器，与 offHeap ExecutionMemoryPool 配合使用。UnsafeMemoryAllocator 实现了 MemoryAllocator 定义的两个方法。

1）allocate：用于分配指定大小（size）的 MemoryBlock。allocate 方法的实现如代码清单 8-14 所示。

代码清单8-14 UnsafeMemoryAllocator的allocate方法

```
@Override
public MemoryBlock allocate(long size) throws OutOfMemoryError {
  long address = Platform.allocateMemory(size); // 在堆外内存分配指定大小的内存
  MemoryBlock memory = new MemoryBlock(null, address, size); // 创建MemoryBlock
  if (MemoryAllocator.MEMORY_DEBUG_FILL_ENABLED) {
    memory.fill(MemoryAllocator.MEMORY_DEBUG_FILL_CLEAN_VALUE);
  }
  return memory; // 返回MemoryBlock
}
```



根据代码清单 8-14，UnsafeMemoryAllocator 的 allocate 方法的执行步骤如下。

① 在堆外内存分配指定大小的内存。Platform 的 allocateMemory 方法实际代理了 sun.misc.Unsafe 的 allocateMemory 方法，sun.misc.Unsafe 的 allocateMemory 方法将返回分配的内存地址。

② 创建 MemoryBlock，可以看到未传递 MemoryBlock 的 obj 属性。

③ 返回 MemoryBlock。

2）free：用于释放 MemoryBlock。free 方法的实现如代码清单 8-15 所示。

代码清单8-15 UnsafeMemoryAllocator的free方法

```
@Override
public void free(MemoryBlock memory) {
  assert (memory.obj == null) :
    "baseObject not null; are you trying to use the off-heap allocator to free on-
        heap memory?";
  if (MemoryAllocator.MEMORY_DEBUG_FILL_ENABLED) {
    memory.fill(MemoryAllocator.MEMORY_DEBUG_FILL_FREED_VALUE);
  }
  Platform.freeMemory(memory.offset);
}
```

根据代码清单 8-15，UnsafeMemoryAllocator 的 free 方法主要是调用了 Platform 的 freeMemory 方法，后者实际代理了 sun.misc.Unsafe 的 freeMemory 方法。

仅供非商业用途或交流学习使用

根据以上分析，UnsafeMemoryAllocator 对内存的分配的确是基于操作系统的内存地址的，图 8-5 很好地表示了 UnsafeMemoryAllocator 的工作原理。

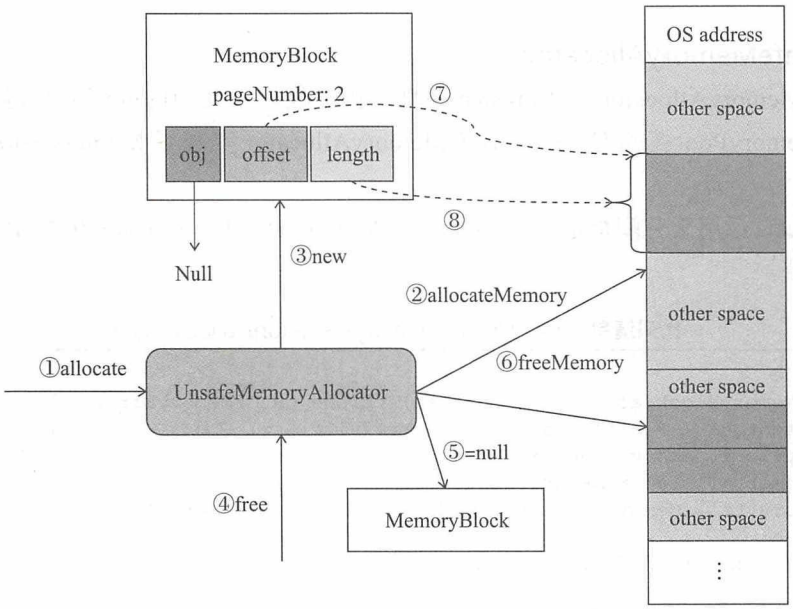


图 8-5 UnsafeMemoryAllocator 的工作原理

这里对图 8-5 进行简单的描述。

标记①：调用 UnsafeMemoryAllocator 的 allocate 方法分配 MemoryBlock。

标记②：UnsafeMemoryAllocator 调用 sun.misc.Unsafe 的 allocateMemory 方法请求操作系统分配内存。

标记③：操作系统分配了内存后，将此块内存的地址信息返回给 UnsafeMemoryAllocator。UnsafeMemoryAllocator 利用内存地址信息和内存大小创建 MemoryBlock，此 MemoryBlock 的 obj 属性为 null。

标记④：调用 UnsafeMemoryAllocator 的 free 方法释放 MemoryBlock。

标记⑤：在调用 UnsafeMemoryAllocator 的 free 方法之前，调用方已经将此 MemoryBlock 的引用设置为 null。

标记⑥：UnsafeMemoryAllocator 调用 sun.misc.Unsafe 的 freeMemory 方法请求操作系统释放内存。

标记⑦：MemoryBlock 的 offset 属性保存了 Page 在操作系统内存中的地址。

标记⑧：MemoryBlock 的 length 属性保存了 Page 的页面大小（即从 offset 开始，连续内存空间的大小）。

仅供非商业用途或交流学习使用

8.4 任务内存管理器

任务内存管理器（TaskMemoryManager）用于管理单个任务尝试的内存分配与释放。TaskMemoryManager 实际上依赖于 MemoryManager 提供的内存管理能力，多个 TaskMemoryManager 将共享 MemoryManager 所管理的内存。一次任务尝试有很多组件需要使用内存，这些组件都借助于 TaskMemoryManager 提供的服务对实际的物理内存进行消费，它们统称为内存消费者（MemoryConsumer）。本节首先详细分析 TaskMemoryManager，然后介绍 MemoryConsumer。

8.4.1 TaskMemoryManager 详解

TaskMemoryManager 包含以下成员。

- PAGE_NUMBER_BITS：用于寻址 Page 表的位数。静态常量 PAGE_NUMBER_BITS 的值为 13。在 64 位的长整型中将使用高位的 13 位存储页号。
- OFFSET_BITS：用于保存编码后的偏移量的位数。静态常量 OFFSET_BITS 的值为 51。在 64 位的长整型中将使用低位的 51 位存储偏移量。
- PAGE_TABLE_SIZE：Page 表中的 Page 数量。静态常量 PAGE_TABLE_SIZE 的值为 8192，实际是将 1 向左位移 13（即 PAGE_NUMBER_BITS）位所得的值。
- MAXIMUM_PAGE_SIZE_BYTES：最大的 Page 大小。静态常量 MAXIMUM_PAGE_SIZE_BYTES 的值为 17 179 869 176，即（2^32–1）×8。
- MASK_LONG_LOWER_51_BITS：长整型的低 51 位的位掩码。静态常量 MASK_LONG_LOWER_51_BITS 的值为 2 251 799 813 685 247，即十六进制 0x7FFFFF-FFFFFFFL。
- pageTable：Page 表。pageTable 实际为 Page（即 MemoryBlock）的数组，数组长度为 PAGE_TABLE_SIZE。
- allocatedPages：用于跟踪空闲 Page 的 BitSet。
- memoryManager：即 MemoryManager。
- taskAttemptId：TaskMemoryManager 所管理任务尝试的身份标识。
- tungstenMemoryMode：Tungsten 的内存模式。TaskMemoryManager 的 getTungstenMemoryMode() 方法专门用于返回 tungstenMemoryMode 的值。
- consumers：用于跟踪可溢出的内存消费者（MemoryConsumer）。
- acquiredButNotUsed：任务尝试已经获得但是并未使用的内存大小。

TaskMemoryManager 提供了很多方法，下面逐一介绍。

1. acquireExecutionMemory

acquireExecutionMemory 方法（见代码清单 8-16）用于为内存消费者获得指定大小（单位为字节）的内存。当 Task 没有足够的内存时，将调用 MemoryConsumer 的 spill 方法（见代码清单 8-27）释放内存。

仅供非商业用途或交流学习使用

代码清单8-16 获取执行内存

```
public long acquireExecutionMemory(long required, MemoryConsumer consumer) {
  assert(required >= 0);
  assert(consumer != null);
  MemoryMode mode = consumer.getMode();
  synchronized (this) { // 为当前的任务尝试按照指定的存储模式获取指定大小的内存
    long got = memoryManager.acquireExecutionMemory(required, taskAttemptId, mode);
    if (got < required) {
      for (MemoryConsumer c: consumers) {
        if (c != consumer && c.getUsed() > 0 && c.getMode() == mode) {
          try {
            long released = c.spill(required - got, consumer); // 为当前的任务尝试腾
                                                               // 出内存
            if (released > 0) {
              logger.debug("Task {} released {} from {} for {}", taskAttemptId,
                Utils.bytesToString(released), c, consumer);
              got += memoryManager.acquireExecutionMemory(required - got, task
                  AttemptId, mode);
              if (got >= required) { // 已经获得的内存达到期望的内存大小
                break;
              }
            }
          } catch (IOException e) {
            logger.error("error while calling spill() on " + c, e);
            throw new OutOfMemoryError("error while calling spill() on " + c + " : "
              + e.getMessage());
          }
        }
      }
    }

    if (got < required) { // 已经获得的内存未达到期望的内存大小
      try {
        long released = consumer.spill(required - got, consumer); // 将数据溢出到磁
                                                                  // 盘以释放内存
        if (released > 0) { // 获取内存的最后尝试
          logger.debug("Task {} released {} from itself ({})", taskAttemptId,
            Utils.bytesToString(released), consumer);
          got += memoryManager.acquireExecutionMemory(required - got, task
              AttemptId, mode);
        }
      } catch (IOException e) {
        logger.error("error while calling spill() on " + consumer, e);
        throw new OutOfMemoryError("error while calling spill() on " + consumer
            + " : "
          + e.getMessage());
      }
    }

    consumers.add(consumer); // 将当前申请获得内存的MemoryConsumer添加到consumers中
    logger.debug("Task {} acquired {} for {}", taskAttemptId, Utils.bytes ToString
        (got), consumer);
    return got; // 返回最终获得的内存大小
  }
}
```



根据代码清单 8-16，acquireExecutionMemory 方法的执行步骤如下：

1）调用 MemoryManager 的 acquireExecutionMemory 方法尝试为当前任务按指定的存储模式获取指定大小的内存。

2）如果逻辑上已经获得的内存未达到期望的内存大小，那么遍历 consumers 中与指定内存模式相同且已经使用了内存的 MemoryConsumer（不包括当前申请内存的 MemoryConsumer），对每个 MemoryConsumer 执行如下操作。

① 调用 MemoryConsumer 的 spill 方法（见代码清单 8-27）溢出数据到磁盘，以释放内存。

② 如果 MemoryConsumer 释放了内存空间，那么调用 MemoryManager 的 acquireExecutionMemory 方法尝试为当前任务按指定的存储模式继续获取期望获得的内存与已经获得的内存之间差值大小的内存。

③ 如果已经获得的内存达到了期望，则进入 3），否则继续 2）。

3）如果已经获得的内存未达到期望的内存大小，则调用当前 MemoryConsumer 的 pill 方法溢出数据到磁盘以释放内存，并且调用 MemoryManager 的 acquireExecutionMemory 方法做最后的尝试。

4）将当前申请获得内存的 MemoryConsumer 添加到 consumers 中，并返回最终获得的内存大小。

2. releaseExecutionMemory

releaseExecutionMemory 方法（见代码清单 8-17）用于为内存消费者释放指定大小（单位为字节）的内存。

代码清单8-17 释放执行内存

```
public void releaseExecutionMemory(long size, MemoryConsumer consumer) {
  logger.debug("Task {} release {} from {}", taskAttemptId, Utils.bytesToString
      (size), consumer);
  memoryManager.releaseExecutionMemory(size, taskAttemptId, consumer.getMode());
}
```

根据代码清单 8-17，releaseExecutionMemory 方法实际调用了 MemoryManager 的 releaseExecutionMemory 方法。

3. showMemoryUsage

showMemoryUsage 方法（见代码清单 8-18）用于将任务尝试、各个 MemoryConsumer 及 MemoryManager 管理的执行内存和存储内存的使用情况打印到日志。

代码清单8-18 展示内存使用信息

```
public void showMemoryUsage() {
  logger.info("Memory used in task " + taskAttemptId);
  synchronized (this) {
    long memoryAccountedForByConsumers = 0;
```

```
    for (MemoryConsumer c: consumers) {
      long totalMemUsage = c.getUsed();
      memoryAccountedForByConsumers += totalMemUsage;
      if (totalMemUsage > 0) {
        logger.info("Acquired by " + c + ": " + Utils.bytesToString(totalMemUsage));
      }
    }
    long memoryNotAccountedFor =
      memoryManager.getExecutionMemoryUsageForTask(taskAttemptId) - memory Accounted
ForByConsumers;
    logger.info(
      "{} bytes of memory were used by task {} but are not associated with specific
          consumers",
      memoryNotAccountedFor, taskAttemptId);
    logger.info(
      "{} bytes of memory are used for execution and {} bytes of memory are used
          for storage",
      memoryManager.executionMemoryUsed(), memoryManager.storageMemoryUsed());
  }
}
```

4. pageSizeBytes

pageSizeBytes 方法（见代码清单 8-19）用于获得 Page 的大小（单位为字节）。其实际为 MemoryManager 的 pageSizeBytes（见代码清单 8-10）属性。

代码清单8-19　获取Page的大小

```
public long pageSizeBytes() {
  return memoryManager.pageSizeBytes();
}
```

5. allocatePage

allocatePage 方法（见代码清单 8-20）用于给 MemoryConsumer 分配指定大小（单位为字节）的 MemoryBlock。

代码清单8-20　分配Page

```
public MemoryBlock allocatePage(long size, MemoryConsumer consumer) {
  assert(consumer != null);
  assert(consumer.getMode() == tungstenMemoryMode);
  if (size > MAXIMUM_PAGE_SIZE_BYTES) { // 请求获得的页大小不能超出限制
    throw new IllegalArgumentException(
      "Cannot allocate a page with more than " + MAXIMUM_PAGE_SIZE_BYTES + " bytes");
  }

  long acquired = acquireExecutionMemory(size, consumer); // 获取逻辑内存
  if (acquired <= 0) {
    return null;
  }

  final int pageNumber;
  synchronized (this) {
    pageNumber = allocatedPages.nextClearBit(0); // 获得还未分配的页号
```

```
    if (pageNumber >= PAGE_TABLE_SIZE) {
      releaseExecutionMemory(acquired, consumer);
      throw new IllegalStateException(
        "Have already allocated a maximum of " + PAGE_TABLE_SIZE + " pages");
    }
    allocatedPages.set(pageNumber); // 将此页号记为已分配
  }
  MemoryBlock page = null;
  try {
    page = memoryManager.tungstenMemoryAllocator().allocate(acquired); // 分配指定
                                                                       // 大小的MemoryBlock
  } catch (OutOfMemoryError e) {
    logger.warn("Failed to allocate a page ({} bytes), try again.", acquired);
    synchronized (this) {
      acquiredButNotUsed += acquired;
      allocatedPages.clear(pageNumber);
    }
    return allocatePage(size, consumer);
  }
  page.pageNumber = pageNumber; // 给MemoryBlock指定页号
  pageTable[pageNumber] = page; // 将页号（pageNumber）与MemoryBlock之间的对应关系放
                                // 入pageTable中
  if (logger.isTraceEnabled()) {
    logger.trace("Allocate page number {} ({} bytes)", pageNumber, acquired);
  }
  return page; // 返回MemoryBlock
}
```

根据代码清单 8-20，allocatePage 方法的执行步骤如下。

1）对输入参数进行校验。

2）调用 acquireExecutionMemory 方法（见代码清单 8-16）获取逻辑内存。如果获取到的内存大小小于等于零，那么返回 null。

3）从 allocatedPages 获得还未分配的页号，将此页号记为已分配。

4）获取 Tungsten 采用的内存分配器（MemoryAllocator），并调用 MemoryAllocator 的 allocate 方法分配指定大小的 MemoryBlock（即获取物理内存）。如果 allocate 方法抛出了 OutOfMemoryError，那么说明物理内存大小小于 MemoryManager 认为自己管理的逻辑内存大小，此时需要更新 acquiredButNotUsed，从 allocatedPages 中清除此页号并再次调用 allocatePage 方法。

5）给 MemoryBlock 指定页号，并将页号（pageNumber）与 MemoryBlock 之间的对应关系放入 pageTable 中。

6）返回 MemoryBlock。



注意 根据 allocatePage 方法的实现，我们看到 TaskMemoryManager 在分配 Page 时，首先从指定内存模式对应的 ExecutionMemoryPool 中申请获得逻辑内存，然后会选择内存模式对应的 MemoryAllocator 申请获得物理内存。

6. freePage

freePage 方法（见代码清单 8-21）用于释放给 MemoryConsumer 分配的 MemoryBlock。

代码清单8-21 释放Page

```
public void freePage(MemoryBlock page, MemoryConsumer consumer) {
  assert (page.pageNumber != -1) :
    "Called freePage() on memory that wasn't allocated with allocatePage()";
  assert(allocatedPages.get(page.pageNumber));
  pageTable[page.pageNumber] = null; // 清理pageTable中指定页号对应的MemoryBlock
  synchronized (this) {
    allocatedPages.clear(page.pageNumber); // 清空allocatedPages对MemoryBlock的页号
                                           // 的跟踪
  }
  if (logger.isTraceEnabled()) {
    logger.trace("Freed page number {} ({} bytes)", page.pageNumber, page.size());
  }
  long pageSize = page.size();
  memoryManager.tungstenMemoryAllocator().free(page); // 释放MemoryBlock
  releaseExecutionMemory(pageSize, consumer); // 释放MemoryManager管理的逻辑内存
}
```

根据代码清单 8-21，freePage 方法的执行步骤如下。

1）清理 pageTable 中保存的 MemoryBlock。

2）清空 allocatedPages 对 MemoryBlock 的页号的跟踪。

3）获取 MemoryBlock 的页大小。

4）获取 Tungsten 采用的内存分配器（MemoryAllocator），并调用 MemoryAllocator 的 free 方法释放 MemoryBlock（即释放物理内存）。

5）调用 releaseExecutionMemory 方法（见代码清单 8-17）释放 MemoryManager 管理的逻辑内存。



根据 freePage 方法的实现，我们看到 TaskMemoryManager 在释放 Page 时，首先使用内存模式对应的 MemoryAllocator 释放物理内存，然后从指定内存模式对应的 ExecutionMemoryPool 中释放逻辑内存。freePage 与 allocatePage 操作的顺序正好相反。

7. encodePageNumberAndOffset

encodePageNumberAndOffset 方法（见代码清单 8-22）用于根据给定的 Page（即 MemoryBlock）和 Page 中偏移量的地址，返回页号和相对于内存块起始地址的偏移量（64 位长整型）。

代码清单8-22 对页号和偏移量编码

```
public long encodePageNumberAndOffset(MemoryBlock page, long offsetInPage) {
  if (tungstenMemoryMode == MemoryMode.OFF_HEAP) {
    offsetInPage -= page.getBaseOffset(); // 获取页的偏移量
  }
  return encodePageNumberAndOffset(page.pageNumber, offsetInPage);
```

```
}
// 获取页号相对于内存块起始地址的偏移量
@VisibleForTesting
public static long encodePageNumberAndOffset(int pageNumber, long offsetInPage) {
  assert (pageNumber != -1) : "encodePageNumberAndOffset called with invalid page";
  return (((long) pageNumber) << OFFSET_BITS) | (offsetInPage & MASK_LONG_
      LOWER_51_BITS);
}
```

根据代码清单 8-22，encodePageNumberAndOffset 方法的执行步骤如下。

1）如果 Tungsten 的内存模式是堆外内存，此时的参数 offsetInPage 是操作系统内存的绝对地址，offsetInPage 与 MemoryBlock 的起始地址之差就是相对于起始地址的偏移量（64 位地址的低 51 位）。

2）调用重载的静态方法 encodePageNumberAndOffset 通过位运算将页号存储到 64 位长整型的高 13 位中，并将偏移量存储到 64 位长整型的低 51 位中，返回生成的 64 位的长整型。

8. decodePageNumber

decodePageNumber 方法（见代码清单 8-23）用于将 64 位的长整型右移 51 位（只剩下页号），然后转换为整型以获得 Page 的页号。

代码清单8-23 解码页号

```
@VisibleForTesting
public static int decodePageNumber(long pagePlusOffsetAddress) {
  return (int) (pagePlusOffsetAddress >>> OFFSET_BITS);
}
```

9. decodeOffset

decodeOffset 方法（见代码清单 8-24）用于将 64 位的长整型与 51 位的掩码按位进行与运算，以获得在 Page 中的偏移量。

代码清单8-24 解码偏移量

```
private static long decodeOffset(long pagePlusOffsetAddress) {
  return (pagePlusOffsetAddress & MASK_LONG_LOWER_51_BITS);
}
```

10. getPage

getPage 方法（见代码清单 8-25）用于通过 64 位的长整型，获取 Page 在内存中的对象。此方法在 Tungsten 采用堆内存模式时有效，否则返回 null。

代码清单8-25 获取Page

```
public Object getPage(long pagePlusOffsetAddress) {
  if (tungstenMemoryMode == MemoryMode.ON_HEAP) { // Tungsten的内存模式是堆内存
    final int pageNumber = decodePageNumber(pagePlusOffsetAddress); // 获得Page的页号
    assert (pageNumber >= 0 && pageNumber < PAGE_TABLE_SIZE);
```

```
      final MemoryBlock page = pageTable[pageNumber]; // 从pageTable中取出MemoryBlock
      assert (page != null);
      assert (page.getBaseObject() != null);
      return page.getBaseObject(); // 返回MemoryBlock的obj
    } else { // Tungsten的内存模式是堆外内存
      return null; // 由于使用操作系统内存时不需要在JVM堆上创建对象，因此直接返回null
    }
  }
```

根据代码清单 8-25，getPage 方法的执行步骤如下。

1）如果 Tungsten 的内存模式是堆内存，首先调用 decodePageNumber 方法（见代码清单 8-23）获得 Page 的页号，然后从 pageTable 中取出 MemoryBlock，最后返回 MemoryBlock 的 obj。

2）如果 Tungsten 的内存模式是堆外内存，由于使用操作系统内存时不需要在 JVM 堆上创建对象，因此直接返回 null。

11. getOffsetInPage

getOffsetInPage 方法（见代码清单 8-26）用于通过 64 位的长整型，获取在 Page 中的偏移量。

代码清单 8-26

```
public long getOffsetInPage(long pagePlusOffsetAddress) {
  final long offsetInPage = decodeOffset(pagePlusOffsetAddress); // 获得在Page中的偏
      移量
  if (tungstenMemoryMode == MemoryMode.ON_HEAP) { // Tungsten的内存模式是堆内存
    return offsetInPage; // 返回在Page中的偏移量
  } else { // Tungsten的内存模式是堆外内存
    final int pageNumber = decodePageNumber(pagePlusOffsetAddress); // 获得Page的页号
    assert (pageNumber >= 0 && pageNumber < PAGE_TABLE_SIZE);
    final MemoryBlock page = pageTable[pageNumber]; // 从pageTable中获得与页号对应的
        MemoryBlock
    assert (page != null);
    return page.getBaseOffset() + offsetInPage; // 返回Page在操作系统内存中的偏移量
  }
}
```

根据代码清单 8-26，getOffsetInPage 方法的执行步骤如下。

1）调用 decodeOffset 方法（见代码清单 8-24）从 64 位的长整型中解码获得在 Page 中的偏移量。

2）如果 Tungsten 的内存模式是堆内存，则返回第 1）步得到的偏移量。

3）如果 Tungsten 的内存模式是堆外内存，首先调用 decodePageNumber 从 64 位的长整型中解码获得 Page 的页号，然后从 pageTable 中获得与页号对应的 MemoryBlock，最后将 Page 在操作系统内存中的地址与第 1）步得到的偏移量之和作为偏移量。

除了以上方法，还有 cleanUpAllAllocatedMemory（用于清空所有 Page 和内存的方法）、getMemoryConsumptionForThisTask（用于获取任务尝试消费的所有内存的大小）、

getTungstenMemoryMode（用于返回 tungstenMemoryMode 的值）等方法。即使不分析这些方法，也不会妨碍读者对 Tungsten 的理解，因此留给感兴趣的读者自行阅读。

8.4.2 内存消费者

抽象类 MemoryConsumer 定义了内存消费者的规范，它通过 TaskMemoryManager 在执行内存（堆内存或堆外内存）上申请或释放内存。

MemoryConsumer 是抽象类，其抽象方法 spill 需要子类去实现，Spark 中有很多 MemoryConsumer 的子类，如图 8-6 所示。

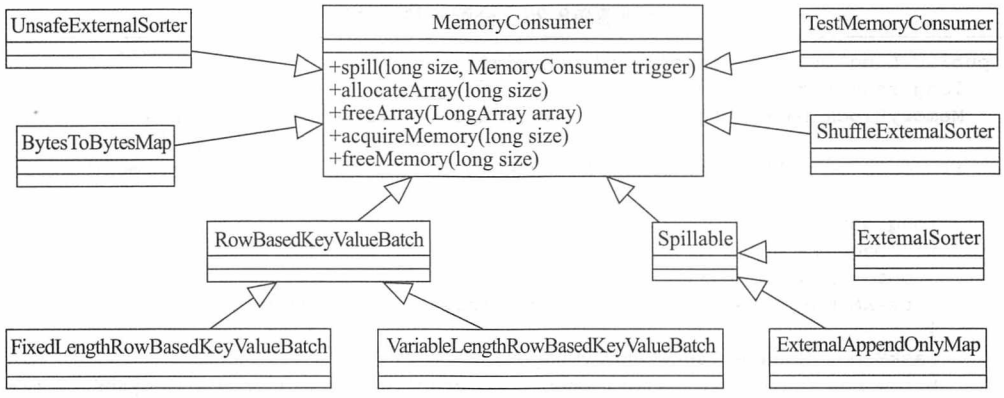


图 8-6 MemoryConsumer 的继承体系

图 8-6 中的 TestMemoryConsumer 用于测试，其他的实现类都将是 TaskMemoryManager 所管理内存的消费者。

MemoryConsumer 有以下属性。

- taskMemoryManager：即 TaskMemoryManager。
- pageSize：MemoryConsumer 要消费的 Page 的大小。
- mode：即内存模式（MemoryMode）。MemoryConsumer 中提供了 getMode 方法获取 mode。
- used：当前消费者已经使用的执行内存的大小。MemoryConsumer 中提供了 getUsed 方法获取 used。

MemoryConsumer 提供了很多方法，下面逐一介绍。

1. 抽象方法 spill

MemoryConsumer 中定义了需要子类实现的抽象方法 spill，当任务尝试没有足够的内存可用时，TaskMemoryManager 将调用此方法把一些数据溢出到磁盘，以释放内存。抽象方法 spill 的定义如下。

```
public abstract long spill(long size, MemoryConsumer trigger) throws IOException;
```

2. 模板方法 spill

spill 方法（见代码清单 8-27）用于调用子类实现的 spill 方法将数据溢出到磁盘。

代码清单8-27 模板方法spill

```
public void spill() throws IOException {
  spill(Long.MAX_VALUE, this);
}
```

3. allocateArray

allocateArray 方法（见代码清单 8-28）用于分配指定大小的长整型数组。

代码清单8-28 分配长整型数组

```
public LongArray allocateArray(long size) {
  long required = size * 8L; // 计算所需的Page大小
  MemoryBlock page = taskMemoryManager.allocatePage(required, this); // 分配指定大
                                                                     // 小的MemoryBlock
  if (page == null || page.size() < required) { // 分配得到的MemoryBlock的大小小于所需
                                                // 的大小
    long got = 0;
    if (page != null) {
      got = page.size();
      taskMemoryManager.freePage(page, this); // 释放MemoryBlock
    }
    taskMemoryManager.showMemoryUsage();
    throw new OutOfMemoryError("Unable to acquire " + required + " bytes of memory,
        got " + got);
  }
  used += required; // 将required累加到used，即更新已经使用的内存大小
  return new LongArray(page); // 创建并返回LongArray
}
```

根据代码清单 8-28，allocateArray 方法的执行步骤如下。

1）计算所需的 Page 大小（即 required）。由于长整型占用 8 个字节，所以需要乘以 8。

2）调用 TaskMemoryManager 的 allocatePage 方法（见代码清单 8-20），给当前 MemoryConsumer 分配指定大小的 Page（即 MemoryBlock）。

3）如果分配得到的 MemoryBlock 的大小小于所需的大小 required，则调用 TaskMemoryManager 的 freePage 方法（见代码清单 8-21）释放 MemoryBlock，然后调用 TaskMemoryManager 的 showMemoryUsage 方法（见代码清单 8-18）打印内存使用信息并抛出 OutOfMemoryError。

4）如果分配到的 MemoryBlock 的大小大于等于所需的 required，则将 required 累加到 used，然后创建并返回 LongArray。



注意 LongArray 并非使用了 Java 的 new Long[size] 语法来创建长整型数组，而是调用 sun.misc.Unsafe 的 putLong(object, offset, value) 和 getLong(object, offset) 两个方法来模拟实现长整型数组。

4. freeArray

freeArray 方法（见代码清单 8-29）用于释放长整型数组。

代码清单8-29 释放长整型数组

```
public void freeArray(LongArray array) {
  freePage(array.memoryBlock());
}
protected void freePage(MemoryBlock page) {
  used -= page.size();
  taskMemoryManager.freePage(page, this);
}
```

根据代码清单 8-29，freeArray 方法调用了 freePage 方法，而 freePage 方法首先更新 used，然后调用 TaskMemoryManager 的 freePage 方法（见代码清单 8-21）释放 MemoryBlock。

5. acquireMemory

acquireMemory 方法（见代码清单 8-30）用于获得指定大小（单位为字节）的内存。

代码清单8-30 获取内存

```
public long acquireMemory(long size) {
  long granted = taskMemoryManager.acquireExecutionMemory(size, this);
  used += granted;
  return granted;
}
```

根据代码清单 8-30，acquireMemory 方法首先调用 TaskMemoryManager 的 acquireExecutionMemory 方法（见代码清单 8-16）获取指定大小的内存，然后更新 used，最后返回实际获得的内存大小（即 granted）。

6. freeMemory

freeMemory 方法（见代码清单 8-31）用于释放指定大小（单位为字节）的内存。

代码清单8-31 释放内存

```
public void freeMemory(long size) {
  taskMemoryManager.releaseExecutionMemory(size, this);
  used -= size;
}
```

根据代码清单 8-31，freeMemory 方法首先调用 TaskMemoryManager 的 releaseExecutionMemory 方法（见代码清单 8-17）释放指定大小的内存，然后更新 used。

8.4.3 执行内存整体架构

通过对内存管理器（MemoryManager）、执行内存池（ExecutionMemoryPool）、Tungsten、内存分配器（MemoryAllocator）、任务内存管理器（TaskMemoryManager）、内存消费者（Memory-

Consumer）等组件的分析和总结，现在我们可以通过图 8-7 来说明执行内存的整体架构。

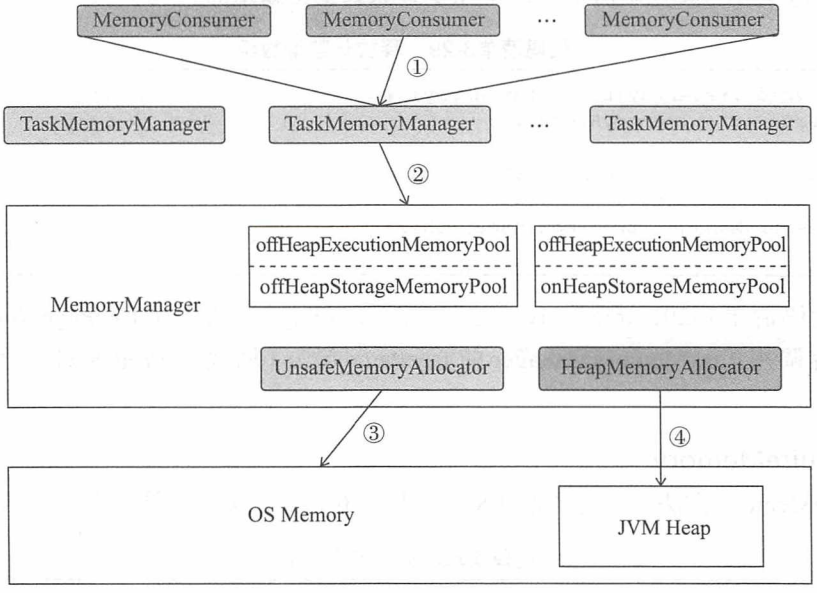


图 8-7　执行内存的整体架构

图 8-7 中，从下往上展示了 Spark 执行内存的各个组成部分，上层的组件依赖于下层提供的服务或支持。操作系统内存（OS Memory）是整个架构的基础，无论执行内存如何分配，都离不开系统内存的支持。Java 虚拟机（JVM）的堆内存（Heap）提供了对 Java 对象的存储支持，其实质依然是从操作系统申请获得的内存。内存管理器（MemoryManager）提供了四种逻辑上的内存池，分别为堆外执行内存池（offHeapExecutionMemoryPool）、堆上执行内存池（onHeapExecutionMemoryPool）、堆外存储内存池（offHeapStorageMemoryPool）、堆上存储内存池（onHeapStorageMemoryPool）。内存管理器提供了在 Tungsten 的堆外内存上分配内存的 UnsafeMemoryAllocator 和在 Tungsten 的堆内存上分配内存的 HeapMemoryAllocator。UnsafeMemoryAllocator 通过 sun.misc.Unsafe 的各种 API 操纵操作系统内存，HeapMemoryAllocator 则通过在 JVM Heap 上分配对象的方式操纵 JVM Heap。由于每个节点只有一个 MemoryManager，而每个任务尝试都会有一个 TaskMemoryManager 为其管理内存，所以多个 TaskMemoryManager 将分享 MemoryManager 管理的内存。每个 TaskMemoryManager 管理的任务内存又会有多个内存消费者（MemoryConsumer）进行消费。

小贴士：图 8-7 中将存储内存池也展示出来，有两个原因：一是存储内存和执行内存的“软”边界将导致存储内存在运行期可能转变为执行内存，加入到执行内存的体系中；二是为了说明存储内存池也离不开操作系统内存和 JVM 堆内存的支持。

有了对执行内存的整体架构的了解，这里对图 8-7 中的记号进行说明。

标记①：MemoryConsumer 调用 TaskMemoryManager 提供的 API，获取 / 释放执行内存。

标记②：TaskMemoryManager 提供的 API 实际都依赖于 MemoryManager 的具体实现。MemoryManager 通过 offHeapExecutionMemoryPool 和 onHeapExecutionMemoryPool，分别对堆外内存和堆内存进行逻辑操作。MemoryManager 通过 UnsafeMemoryAllocator 和 HeapMemoryAllocator，分别对堆外内存和堆内存进行物理操作。

标记③：UnsafeMemoryAllocator 通过 sun.misc.Unsafe 的各种 API 操作系统内存。

标记④：HeapMemoryAllocator 通过在 JVM Heap 上分配对象的方式操纵 JVM Heap。

8.5 Task 详解

本书在 7.4.9 节曾经介绍过 Task 的创建和提交，DAGScheduler 虽然负责创建和提交 Task，TaskScheduler 也会对 Task 进行资源调度与分配，但是它们却不关心 Task 的实现细节。Task 的实现与计算引擎息息相关，所以本节将详细介绍它。

8.5.1 任务上下文 TaskContext

TaskContext 维护了 Task 执行时的上下文信息，所以我们需要对 TaskContext 的功能进行分析。抽象类 TaskContext 中定义了一系列抽象方法，由于 TaskContext 只有一个实现类 TaskContextImpl，所以我们将直接介绍 TaskContextImpl。TaskContext 还有一个利用 ThreadLocal 技术的伴生对象，用于维护每个 Task 线程的 TaskContextImpl。

1. TaskContextImpl 详解

TaskContextImpl 是抽象类 TaskContext 的唯一实现，我们从了解它的属性信息开始，逐步深入 TaskContextImpl 的实现。

TaskContextImpl 的属性信息如下。

1）stageId：Task 所属 Stage 的身份标识。

2）partitionId：Task 对应的分区索引。

3）taskAttemptId：任务尝试的身份标识。

4）attemptNumber：任务尝试号。

5）taskMemoryManager：Task 内存管理器 TaskMemoryManager。

6）metricsSystem：即度量系统 MetricsSystem。

7）taskMetrics：用于跟踪 Task 执行过程的度量信息，类型为 TaskMetrics。

8）onCompleteCallbacks：保存任务执行完成后需要回调的 TaskCompletionListener 的数组。

9）onFailureCallbacks：保存任务执行失败后需要回调的 TaskFailureListener 的数组。

10）interrupted：TaskContextImpl 相对应的任务尝试是否已经被 kill 的状态。之所以用 interrupted 作为任务尝试被 kill 的状态变量，是因为 kill 实际是通过对执行任务尝试的线程

进行中断实现的。

11）completed：TaskContextImpl 相对应的 Task 是否已经完成的状态。

12）failed：TaskContextImpl 相对应的 Task 是否已经失败的状态。

TaskContextImpl 重写了 TaskContext 的所有方法，下面一一进行介绍。

13）addTaskCompletionListener：用于向 onCompleteCallbacks 中添加 TaskCompletionListener，其实现如代码清单 8-32 所示。

14）addTaskFailureListener：用于向 onFailureCallbacks 中添加 TaskFailureListener，其实现如代码清单 8-32 所示。

代码清单8-32 添加任务完成或失败的监听器

```
override def addTaskCompletionListener(listener: TaskCompletionListener): this.type = {
  onCompleteCallbacks += listener
  this
}
override def addTaskFailureListener(listener: TaskFailureListener): this.type = {
  onFailureCallbacks += listener
  this
}
```

15）markTaskFailed：标记 Task 执行失败，其实现如代码清单 8-33 所示。

代码清单8-33 标记Task执行失败

```
private[spark] def markTaskFailed(error: Throwable): Unit = {
  if (failed) return // 判断Task是否已经被标记为失败。如果是，则返回
  failed = true // 将Task标记为失败
  val errorMsgs = new ArrayBuffer[String](2)
  onFailureCallbacks.reverse.foreach { listener =>
    try {
      listener.onTaskFailure(this, error) // 将错误信息交给任务失败监听器处理
    } catch {
      case e: Throwable =>
        errorMsgs += e.getMessage
        logError("Error in TaskFailureListener", e)
    }
  }
  if (errorMsgs.nonEmpty) {
    throw new TaskCompletionListenerException(errorMsgs, Option(error))
  }
}
```

根据代码清单 8-33，markTaskFailed 的执行步骤如下。

① 判断 Task 是否已经被标记为失败，如果 failed 为 true，则直接返回，否则进入下一步。

② 设置 failed 为 true。

③ 对 onFailureCallbacks 进行反向排序后，对 onFailureCallbacks 中的每一个 TaskFailureListener，调用其 onTaskFailure 方法。如果调用 onTaskFailure 方法的过程中发生了异常，这些异常将被收集到 errorMsgs 中。

④如果 errorMsgs 不为空，则对外抛出携带 errorMsgs 的 TaskCompletionListenerException。

16）markTaskCompleted：标记 Task 执行完成，其实现如代码清单 8-34 所示。

代码清单8-34 标记Task执行完成

```
private[spark] def markTaskCompleted(): Unit = {
  completed = true // 将任务标记为已完成
  val errorMsgs = new ArrayBuffer[String](2)
  onCompleteCallbacks.reverse.foreach { listener =>
    try {
      listener.onTaskCompletion(this) // 将任务完成的消息交给任务完成监听器
    } catch {
      case e: Throwable =>
        errorMsgs += e.getMessage
        logError("Error in TaskCompletionListener", e)
    }
  }
  if (errorMsgs.nonEmpty) {
    throw new TaskCompletionListenerException(errorMsgs)
  }
}
```

根据代码清单 8-34，markTaskCompleted 方法的执行步骤如下。

① 设置 completed 为 true。

② 对 onCompleteCallbacks 进行反向排序后，对 onCompleteCallbacks 中的每一个 TaskCompletionListener，调用其 onTaskCompletion 方法。如果调用 onTaskCompletion 方法的过程中发生了异常，这些异常将被收集到 errorMsgs 中。

③ 如果 errorMsgs 不为空，则对外抛出携带 errorMsgs 的 TaskCompletionListenerException。

17）markInterrupted：标记 Task 已经被 kill，其实现如下。

```
private[spark] def markInterrupted(): Unit = {
  interrupted = true
}
```

18）isCompleted：Task 是否已经完成，此方法实际返回了 completed 属性的值。

19）isRunningLocally：Task 是否在本地运行，此方法固定返回 false。

20）isInterrupted：Task 是否已经被 kill，此方法实际返回了 interrupted 属性。

21）getLocalProperty：获取指定 key 对应的本地属性值，其实现如下。

```
override def getLocalProperty(key: String): String = localProperties.getProperty(key)
```

22）getMetricsSources：通过 Source 名称从 MetricsSystem 中获取 Source 序列，其实现如下。

```
override def getMetricsSources(sourceName: String): Seq[Source] =
  metricsSystem.getSourcesByName(sourceName)
```

23）registerAccumulator：向 TaskMetrics 注册累加器，其实现如下。

```
private[spark] override def registerAccumulator(a: AccumulatorV2[_, _]): Unit = {
  taskMetrics.registerAccumulator(a)
}
```

2. TaskContext 的伴生对象

TaskContext 的伴生对象提供了将 TaskContext 保存到 ThreadLocal 中，用于保证每个任务尝试线程的 TaskContextImpl 的线程安全性。TaskContext 的实现如代码清单 8-35 所示。

代码清单8-35 TaskContext的伴生对象

```
object TaskContext {
  def get(): TaskContext = taskContext.get
  def getPartitionId(): Int = {
    val tc = taskContext.get()
    if (tc eq null) {
      0
    } else {
      tc.partitionId()
    }
  }
  private[this] val taskContext: ThreadLocal[TaskContext] = new ThreadLocal
      [TaskContext]
  protected[spark] def setTaskContext(tc: TaskContext): Unit = taskContext.set(tc)
  protected[spark] def unset(): Unit = taskContext.remove()
  private[spark] def empty(): TaskContextImpl = {
    new TaskContextImpl(0, 0, 0, 0, null, new Properties, null)
  }
}
```

根据代码清单 8-35，TaskContext 中提供的方法如下。

1）get：从 ThreadLocal 中获取当前任务尝试线程的 TaskContextImpl。

2）getPartitionId：从 ThreadLocal 中获取当前任务尝试线程的 TaskContextImpl，然后调用 TaskContextImpl 的 partitionId 方法获取当前 Task 对应的分区索引。

3）setTaskContext：将 TaskContextImpl 设置到 ThreadLocal 中。

4）unset：移除 ThreadLocal 中保存的当前任务尝试线程的 TaskContextImpl。

5）empty：创建一个没有任何实际意义的 TaskContextImpl。

8.5.2 Task 的定义

Task 是 Spark 中作业运行的最小单位，为了容错，每个 Task 可能会有一到多次任务尝试。Task 主要包括 ShuffleMapTask 和 ResultTask 两种。每次任务尝试都会申请单独的连续内存，以执行计算。

抽象类 Task 定义了 Spark 中 Task 的规范，我们首先了解 Task 自身的属性，然后介绍 Task 提供的基本方法及需要子类实现的抽象方法，最后分析 Task 定义的模板方法。

Task 包含的属性如下。

1）stageId：Task 所属 Stage 的身份标识。

2）stageAttemptId：Stage 尝试的身份标识。

3）partitionId：Task 对应的分区索引。

4）metrics：用于跟踪 Task 执行过程的度量信息，类型为 TaskMetrics。

5）localProperties：Task 执行所需的属性信息。

6）jobId：Task 所属 Job 的身份标识。

7）appId：Task 所属 Application 的身份标识，即 SparkContext 的 _applicationId 属性。

8）appAttemptId：Task 所属 Application 尝试的身份标识，即 SparkContext 的 _application AttemptId 属性。

9）taskMemoryManager：Task 内存管理器 TaskMemoryManager。

10）epoch ：MapOutputTracker 跟踪的纪元。此属性由 TaskScheduler 设置，用于故障迁移。

11）context ：Task 执行的上下文信息，即 TaskContextImpl。TaskContextImpl 将被设置到 ThreadLocal 中，以保证其线程安全。

12）taskThread：运行任务尝试的线程。

13）_killed：Task 是否被 kill 的状态。

14）_executorDeserializeTime：对 RDD 进行反序列化所花费的时间。

15）_executorDeserializeCpuTime：对 RDD 进行反序列化所花费的 CPU 时间。

Task 中提供的基本方法如下。

16）setTaskMemoryManager：用于设置 Task 的 taskMemoryManager。

17）killed：用于判断任务尝试是否已被“杀死”。此方法实际返回了 Task 的 _killed 属性。

18）executorDeserializeTime：用于获取 Task 的 _executorDeserializeTime 属性。

19）executorDeserializeCpuTime：用于获取 Task 的 _executorDeserializeCpuTime。

20）collectAccumulatorUpdates：收集 Task 使用的累加器的最新值，并更新到 TaskMetrics 中。

21）kill：用于 kill 任务尝试线程，其实现如代码清单 8-36 所示。

代码清单8-36 kill任务尝试线程

```
def kill(interruptThread: Boolean) {
  _killed = true
  if (context != null) {
    context.markInterrupted()
  }
  if (interruptThread && taskThread != null) {
    taskThread.interrupt()
  }
}
```

根据代码清单 8-36，如果 interruptThread 为 false，则只会将 Task 和 TaskContextImpl 标记为已经被 kill。如果 interruptThread 为 true，还会利用 Java 线程的中断机制中断任务尝试线程。

Task 中提供的抽象方法如下。

1）runTask：运行 Task 的接口，定义如下。

```
def runTask(context: TaskContext): T
```

2）preferredLocations：获取当前 Task 偏好的位置信息，定义如下。

```
def preferredLocations: Seq[TaskLocation] = Nil
```

Task 只提供了一个模板方法，此方法是运行 Task 的入口，它的实现如代码清单 8-37 所示。

代码清单8-37 运行Task

```
final def run(
    taskAttemptId: Long,
    attemptNumber: Int,
    metricsSystem: MetricsSystem): T = {
  SparkEnv.get.blockManager.registerTask(taskAttemptId) // 将任务尝试注册到Block
      InfoManager
  context = new TaskContextImpl( // 创建任务尝试的上下文
    stageId,
    partitionId,
    taskAttemptId,
    attemptNumber,
    taskMemoryManager,
    localProperties,
    metricsSystem,
    metrics)
  TaskContext.setTaskContext(context) // 将任务尝试的上下文保存到ThreadLocal中
  taskThread = Thread.currentThread() // 获取运行任务尝试的线程
  if (_killed) { // 如果任务尝试已经被kill，则将任务尝试及其上下文标记为被kill的状态
    kill(interruptThread = false)
  }
  new CallerContext("TASK", appId, appAttemptId, jobId, Option(stageId), Option
      (stageAttemptId),
    // 创建调用者上下文
    Option(taskAttemptId), Option(attemptNumber)).setCurrentContext()
  try {
    runTask(context) // 调用子类实现的runTask方法运行任务尝试
  } catch {
    case e: Throwable =>
      try {
        context.markTaskFailed(e)
      } catch {
        case t: Throwable =>
          e.addSuppressed(t)
      }
      throw e
  } finally {
    context.markTaskCompleted()
    try {
      Utils.tryLogNonFatalError { //释放任务尝试所占用的堆内存和堆外内存
        SparkEnv.get.blockManager.memoryStore.releaseUnrollMemoryForThisTask(Mem
            oryMode.ON_HEAP)
```

```
            SparkEnv.get.blockManager.memoryStore.releaseUnrollMemoryForThisTask(Mem
                oryMode.OFF_HEAP)
            val memoryManager = SparkEnv.get.memoryManager
            memoryManager.synchronized { memoryManager.notifyAll() }
          }
        } finally {
          TaskContext.unset() // 移除ThreadLocal中保存的当前任务尝试线程的上下文
        }
      }
    }
```

根据代码清单 8-37，run 方法的执行步骤如下。

① 调用 BlockManager 的 registerTask 方法（见代码清单 6-79）将任务尝试注册到 BlockInfoManager。

② 创建任务尝试的上下文。

③ 调用 TaskContext 的伴生对象的 setTaskContext 方法将 TaskContextImpl 设置到 ThreadLocal 中。

④ 获取运行任务尝试的线程，并由 taskThread 属性保存。

⑤ 如果任务尝试已经被 kill，则调用 kill 方法（见代码清单 8-36），将任务尝试及 TaskContextImpl 标记为被 kill（因为 interruptThread 为 false）。

⑥ 创建调用者上下文，即 CallerContext（CallerContext 是 Utils 工具类中提供的保存调用者上下文信息的类型，其具体实现可以参阅附录 A）。

⑦ 调用子类实现的 runTask 方法运行任务尝试。如果执行 runTask 方法时捕获到任何错误，则调用 TaskContextImpl 的 markTaskFailed 方法（见代码清单 8-33），执行所有 TaskFailureListener 的 onTaskFailure 方法。

⑧ 无论任务尝试是否成功，最后在 finally 块中调用 TaskContextImpl 的 markTaskCompleted 方法（见代码清单 8-34），执行所有 TaskCompletionListener 的 onTaskCompletion 方法。

⑨ 在 finally 块中首先调用 MemoryStore 的 releaseUnrollMemoryForThisTask 方法（见代码清单 6-51），释放任务尝试所占用的堆内存和堆外内存，以便唤醒任何等待 MemoryManager 管理的执行内存的任务尝试，然后调用 TaskContext 伴生对象的 unset 方法（见代码清单 8-35），移除 ThreadLocal 中保存的当前 Task 线程的 TaskContextImpl。

8.5.3 ShuffleMapTask 的实现

ShuffleMapTask 类似于 Hadoop 中的 MapTask，从其命名可以看出，它对输入数据计算后，将输出的数据在 Shuffle 之前映射到不同的分区，那么下游处理各个分区的 Task 将知道处理哪些数据。ShuffleMapTask 继承了 Task，并实现了具体的 runTask 方法，其实现如下。

```
override def runTask(context: TaskContext): MapStatus = {
  val threadMXBean = ManagementFactory.getThreadMXBean
```

```
val deserializeStartTime = System.currentTimeMillis()
val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
  threadMXBean.getCurrentThreadCpuTime
} else 0L
val ser = SparkEnv.get.closureSerializer.newInstance()
//对任务进行反序列化
val (rdd, dep) = ser.deserialize[(RDD[_], ShuffleDependency[_, _, _])](
  ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)
_executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime
_executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
  threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime
} else 0L
var writer: ShuffleWriter[Any, Any] = null
try { // 将计算的中间结果写入磁盘文件
  val manager = SparkEnv.get.shuffleManager
  writer = manager.getWriter[Any, Any](dep.shuffleHandle, partitionId, context)
  writer.write(rdd.iterator(partition, context).asInstanceOf[Iterator[_ <: Product2
      [Any, Any]]])
  writer.stop(success = true).get
} catch {
  case e: Exception =>
    try {
      if (writer != null) {
        writer.stop(success = false)
      }
    } catch {
      case e: Exception =>
        log.debug("Could not stop writer", e)
    }
    throw e
 }
}
```

根据 ShuffleMapTask 的 runTask 方法的实现，其执行步骤如下。

1）对任务进行反序列化，得到 RDD 和 ShuffleDependency。第 7 章在介绍 resourceOffer 方法（见代码清单 7-73）的时候，其中对任务的 RDD 和 ShuffleDependency 进行过序列化操作，所以这里需要反序列化。

2）调用 SortShuffleManager 的 getWriter 方法（见代码清单 8-95），获取对指定分区的数据进行磁盘写操作的 SortShuffleWriter。

3）调用 RDD 的 iterator 方法进行迭代计算。有关迭代计算的内容将在 10.4.4 节介绍。

4）调用 SortShuffleWriter 的 write 方法将计算的中间结果写入磁盘文件。

8.5.4 ResultTask 的实现

ResultTask 类似于 Hadoop 中的 ResultTask，从其命名可以看出，它读取上游 ShuffleMapTask 输出的数据并计算得到最终的结果。ResultTask 继承了 Task 并实现了具体的 runTask 方法，其实现如下。

```
override def runTask(context: TaskContext): U = {
```

```
    val threadMXBean = ManagementFactory.getThreadMXBean
    val deserializeStartTime = System.currentTimeMillis()
    val deserializeStartCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
      threadMXBean.getCurrentThreadCpuTime
    } else 0L
    val ser = SparkEnv.get.closureSerializer.newInstance()
    val (rdd, func) = ser.deserialize[(RDD[T], (TaskContext, Iterator[T]) => U)](
        对Task进行反序列化
      ByteBuffer.wrap(taskBinary.value), Thread.currentThread.getContextClassLoader)
    _executorDeserializeTime = System.currentTimeMillis() - deserializeStartTime
    _executorDeserializeCpuTime = if (threadMXBean.isCurrentThreadCpuTimeSupported) {
      threadMXBean.getCurrentThreadCpuTime - deserializeStartCpuTime
    } else 0L
    func(context, rdd.iterator(partition, context)) // 进行迭代计算和最终的处理
  }
```

根据上述代码，ResultTask 的 runTask 方法的执行步骤如下。

1）对序列化的 Task 进行反序列化，得到 RDD 和要执行的函数。第 7 章在介绍 resourceOffer 方法（见代码清单 7-73）的时候，其中对任务的 RDD 和 ShuffleDependency 进行过序列化操作，所以这里需要反序列化。

2）调用 RDD 的 iterator 方法进行迭代计算。有关迭代计算的内容将在 10.4.4 节介绍。

3）调用函数进行最终的处理。

8.6 IndexShuffleBlockResolver 详解

特质 ShuffleBlockResolver 定义了对 Shuffle Block 进行解析的规范，包括获取 Shuffle 数据文件、获取 Shuffle 索引文件、删除指定的 Shuffle 数据文件和索引文件、生成 Shuffle 索引文件、获取 Shuffle 块的数据等。ShuffleBlockResolver 目前只有 IndexShuffleBlockResolver 这唯一的实现类。IndexShuffleBlockResolver 用于创建和维护 Shuffle Block 与物理文件位置之间的映射关系。

IndexShuffleBlockResolver 中包含的属性如下。

- blockManager：即 SparkEnv 的子组件 BlockManager。
- transportConf：即与 Shuffle 相关的 TransportConf。有了 transportConf，就可以方便地对 Shuffle 客户端传输线程数（spark.shuffle.io.clientThreads 属性）和 Shuffle 服务端传输线程数（spark.shuffle.io.serverThreads 属性）进行读取。TransportConf 已在 3.2.1 节详细介绍过，此处不再赘述。

下面介绍 IndexShuffleBlockResolver 中提供的方法。

1. getDataFile

getDataFile 方法（见代码清单 8-38）用于获取 Shuffle 数据文件。其实质上是调用 BlockManager 的子组件 DiskBlockManager 的 getFile 方法（见代码清单 6-21）获取 ShuffleDataBlockId。

代码清单8-38 获取Shuffle数据文件

```
def getDataFile(shuffleId: Int, mapId: Int): File = {
  blockManager.diskBlockManager.getFile(ShuffleDataBlockId(shuffleId, mapId,
      NOOP_REDUCE_ID))
}
```

2. getIndexFile

getIndexFile 方法（见代码清单 8-39）用于获取 Shuffle 索引文件。其实质上是调用 BlockManager 的子组件 DiskBlockManager 的 getFile 方法（见代码清单 6-21）获取 ShuffleIndexBlockId。

代码清单8-39 获取Shuffle索引文件

```
private def getIndexFile(shuffleId: Int, mapId: Int): File = {
  blockManager.diskBlockManager.getFile(ShuffleIndexBlockId(shuffleId, mapId,
      NOOP_REDUCE_ID))
}
```

3. removeDataByMap

removeDataByMap 方法（见代码清单 8-40）用于删除 Shuffle 过程中包含指定 map 任务输出数据的 Shuffle 数据文件和索引文件。

代码清单8-40 删除map任务的输出数据

```
def removeDataByMap(shuffleId: Int, mapId: Int): Unit = {
  var file = getDataFile(shuffleId, mapId) // 获取指定Shuffle中指定map任务输出的数据文件
  if (file.exists()) {
    if (!file.delete()) { // 删除数据文件
      logWarning(s"Error deleting data ${file.getPath()}")
    }
  }
  file = getIndexFile(shuffleId, mapId) // 获取指定Shuffle中指定map任务输出的索引文件
  if (file.exists()) {
    if (!file.delete()) { // 删除索引文件
      logWarning(s"Error deleting index ${file.getPath()}")
    }
  }
}
```

根据代码清单 8-40，removeDataByMap 方法的执行步骤如下。

1）调用 getDataFile 方法获取指定 Shuffle 中指定 map 任务输出的数据文件，然后删除。

2）调用 getIndexFile 方法获取指定 Shuffle 中指定 map 任务输出的索引文件，然后删除。

4. writeIndexFileAndCommit

writeIndexFileAndCommit 方法（见代码清单 8-41）用于将每个 Block 的偏移量写入索引文件，并在最后增加一个表示输出文件末尾的偏移量。

代码清单8-41 生成索引文件

```
def writeIndexFileAndCommit(
    shuffleId: Int,
    mapId: Int,
    lengths: Array[Long],
    dataTmp: File): Unit = {
  val indexFile = getIndexFile(shuffleId, mapId) // 获取指定Shuffle中指定map任务输出的
      索引文件
  val indexTmp = Utils.tempFileWith(indexFile) // 根据索引文件获取临时索引文件的路径
  try {
    val out = new DataOutputStream(new BufferedOutputStream(new FileOutputStream
        (indexTmp)))
    Utils.tryWithSafeFinally {
      var offset = 0L
      out.writeLong(offset)
      for (length <- lengths) { // 遍历每个Block的长度，并作为偏移量写入临时索引文件
        offset += length
        out.writeLong(offset)
      }
    } {
      out.close()
    }

    // 获取指定Shuffle中指定map任务输出的数据文件
    val dataFile = getDataFile(shuffleId, mapId)
    synchronized { // 对给定的索引文件和数据文件是否匹配进行检查
      val existingLengths = checkIndexAndDataFile(indexFile, dataFile, lengths.
          length)
      if (existingLengths != null) { // 索引文件和数据文件不匹配，所以将临时索引文件和临
                                     // 时数据文件删除
        System.arraycopy(existingLengths, 0, lengths, 0, lengths.length)
        if (dataTmp != null && dataTmp.exists()) {
          dataTmp.delete()
        }
        indexTmp.delete()
      } else { // 索引文件和数据文件匹配，将临时的索引文件和数据文件作为正式的索引文件和数据文件
        if (indexFile.exists()) {
          indexFile.delete()
        }
        if (dataFile.exists()) {
          dataFile.delete()
        }
        if (!indexTmp.renameTo(indexFile)) {
          throw new IOException("fail to rename file " + indexTmp + " to " + indexFile)
        }
        if (dataTmp != null && dataTmp.exists() && !dataTmp.renameTo(dataFile)) {
          throw new IOException("fail to rename file " + dataTmp + " to " + dataFile)
        }
      }
    }
  } finally {
    if (indexTmp.exists() && !indexTmp.delete()) {
      logError(s"Failed to delete temporary index file at ${indexTmp.getAbsolutePath}")
    }
  }
}
```

根据代码清单 8-41，writeIndexFileAndCommit 方法的执行步骤如下。

1）调用 getIndexFile 方法获取指定 Shuffle 中指定 map 任务输出的索引文件（即 indexFile）。

2）根据索引文件，获取临时索引文件的路径。Utils 工具类的 tempFileWith 方法的实现请参阅附录 A。

3）遍历每个 Block 的长度，并作为偏移量写入临时索引文件（即 indexTmp）。

4）调用 getDataFile 方法获取指定 Shuffle 中指定 map 任务输出的数据文件（即 dataFile）。

5）调用 checkIndexAndDataFile 方法（此方法主要通过对数据文件的长度和索引文件偏移量的和进行比较来实现，感兴趣的读者可自行研究），对给定的索引文件和数据文件是否匹配进行检查。checkIndexAndDataFile 方法最后将返回索引文件中各个 partition 的长度数据 existingLengths。

6）如果 existingLengths 不等于 null，则将 existingLengths 更新到调用 writeIndexFileAndCommit 方法时传入的 lengths 数组，并将临时索引文件 indexTmp 和临时数据文件 dataTmp 删除。

7）如果 existingLengths 等于 null，这说明是为 map 任务中间结果输出的第一次成功尝试，因而需要将 indexTmp 重命名为 indexFile，将 dataTmp 重命名为 dataFile。

5. getBlockData

getBlockData 方法（见代码清单 8-42）用于获取指定的 ShuffleBlockId 对应的数据。

代码清单8-42 获取Shuffle的Block

```
override def getBlockData(blockId: ShuffleBlockId): ManagedBuffer = {
  // 获取指定map任务输出的索引文件
  val indexFile = getIndexFile(blockId.shuffleId, blockId.mapId)
  val in = new DataInputStream(new FileInputStream(indexFile)) // 读取索引文件的输入流
  try {
    ByteStreams.skipFully(in, blockId.reduceId * 8) // 跳过与当前reduce任务无关的字节
    val offset = in.readLong()
    val nextOffset = in.readLong()
    new FileSegmentManagedBuffer( // 构造并返回FileSegmentManagedBuffer
      transportConf,
      getDataFile(blockId.shuffleId, blockId.mapId),
      offset,
      nextOffset - offset)
  } finally {
    in.close()
  }
}
```

根据代码清单 8-42，getBlockData 方法的执行步骤如下。

1）调用 getIndexFile 方法获取指定 Shuffle 中指定 map 任务输出的索引文件（即 indexFile）。

2）读取 indexFile 的输入流。

3）由于索引文件中按顺序写入了每个 reduce 任务所需的 map 中间输出数据文件的偏移量，因此跳过与当前 ShuffleBlockId 对应的 reduce 任务无关的字节。

4）读入索引文件中当前 ShuffleBlockId 对应的 reduce 任务所需 map 数据的偏移量和长度，构造并返回 FileSegmentManagedBuffer（FileSegmentManagedBuffer 已在 3.2.5 节有过介绍）。

8.7 采样与估算

Spark 在 Shuffle 阶段，给 map 任务的输出增加了缓存、聚合的数据结构。这些数据结构将使用各种执行内存，为了对这些数据结构的大小进行计算，以便于扩充大小或在没有足够内存时溢出到磁盘，特质 SizeTracker 定义了对集合进行采样和估算的规范。

8.7.1 SizeTracker 的实现分析

让我们从 SizeTracker 的成员开始，一起分析 SizeTracker 的实现。SizeTracker 包括的成员如下。

- SAMPLE_GROWTH_RATE：采样增长的速率。例如，速率为 2 时，分别对在 1，2，4，8，……位置上的元素进行采样。SAMPLE_GROWTH_RATE 的值固定为 1.1。
- samples：样本队列。最后两个样本将被用于估算。
- bytesPerUpdate：平均每次更新的字节数。
- numUpdates：更新操作（包括插入和更新）的总次数。
- nextSampleNum：下次采样时，numUpdates 的值，即 numUpdates 的值增长到与 nextSampleNum 相同时，才会再次采样。

了解了 SizeTracker 的属性，我们就可以更容易理解 SizeTracker 提供的方法了。

1. 采集样本 takeSample

takeSample 方法用于采集样本，其实现如代码清单 8-43 所示。

代码清单8-43 采集样本

```
private def takeSample(): Unit = {
  // 估算集合的大小并作为样本
  samples.enqueue(Sample(SizeEstimator.estimate(this), numUpdates))
  if (samples.size > 2) { // 保留样本队列的最后两个样本
    samples.dequeue()
  }
  val bytesDelta = samples.toList.reverse match {
    case latest :: previous :: tail =>
      (latest.size - previous.size).toDouble / (latest.numUpdates - previous.numUpdates)
    case _ => 0
  }
  bytesPerUpdate = math.max(0, bytesDelta) // 计算每次更新的字节数
  // 机选下次采样的采样号
  nextSampleNum = math.ceil(numUpdates * SAMPLE_GROWTH_RATE).toLong
}
```

根据代码清单 8-43，采集样本的步骤如下。

1）调用 SizeEstimator 的 estimate 方法（此方法留给感兴趣的读者自行阅读）估算集合的大小，并将估算的大小和 numUpdates 作为样本放入队列 samples 中。样本（Sample）的定义如下。

```
case class Sample(size: Long, numUpdates: Long)
```

2）保留队列 samples 中的最后两个样本。

3）使用以下公式计算 bytesPerUpdate 的值。

$$\text{bytesPerUpdate}=\frac{(\text{本次采样大小}-\text{上次采样大小})}{(\text{本次采样编号}-\text{上次采样编号})}$$

4）根据采样增长的速率（SAMPLE_GROWTH_RATE）计算 nextSampleNum 的值（即 numUpdates 的值等于多少时进行下一次采样）。

2. 重置样本 resetSamples

resetSamples 方法用于重置 SizeTracker 采集的样本，其实现如代码清单 8-44 所示。

代码清单8-44　重置样本

```
protected def resetSamples(): Unit = {
  numUpdates = 1
  nextSampleNum = 1
  samples.clear()
  takeSample()
}
```

根据代码清单 8-44，重置样本的步骤如下。

1）将 numUpdates 设置为 1。

2）将 nextSampleNum 设置为 1。

3）清空 samples 中的样本。

4）调用 takeSample 方法采集样本。

3. afterUpdate

afterUpdate 方法（见代码清单 8-45）用于向集合中更新了元素之后进行回调，以触发对集合的采样。

代码清单8-45　afterUpdate的实现

```
protected def afterUpdate(): Unit = {
  numUpdates += 1 //更新numUpdates
  if (nextSampleNum == numUpdates) { //如果nextSampleNum与numUpdates相等，则进行采样
    takeSample()
  }
}
```

根据代码清单 8-45，afterUpdate 方法的执行步骤如下。

1）更新 numUpdates。

2）如果 nextSampleNum 与 numUpdates 相等，则调用 takeSample 方法采样。

4. 估算集合大小 estimateSize

estimateSize 方法（见代码清单 8-46）用于估算集合的当前大小（单位为字节）。

代码清单8-46 估算集合大小

```
def estimateSize(): Long = {
  assert(samples.nonEmpty)
  val extrapolatedDelta = bytesPerUpdate * (numUpdates - samples.last.numUpdates)
  (samples.last.size + extrapolatedDelta).toLong
}
```

根据代码清单 8-46，估算集合大小的步骤如下。

1）使用当前的 numUpdates 与上次采样的 numUpdates 之间的差值，乘以 bytesPerUpdate 作为估计要增加的大小（即 extrapolatedDelta）。

2）将上次采样时的集合大小与 extrapolatedDelta 相加作为估算的集合大小。

8.7.2 SizeTracker 的工作原理

经过对 SizeTracker 的实现分析，本节来一起总结 SizeTracker 的工作原理。为便于说明，本节假设采样增长的速率（SAMPLE_GROWTH_RATE）的值不是 1.1，而是 2。我们将会针对初始估算大小为 100 的集合进行 4 次采样。为了便于计算，添加到集合中的元素都只选择 10、20、30 这样的整数。本节假设 SizeEstimator 的 estimate 方法对集合大小的估算非常精确，即与集合的实际大小一致。SizeTracker 的工作原理如图 8-8 所示。

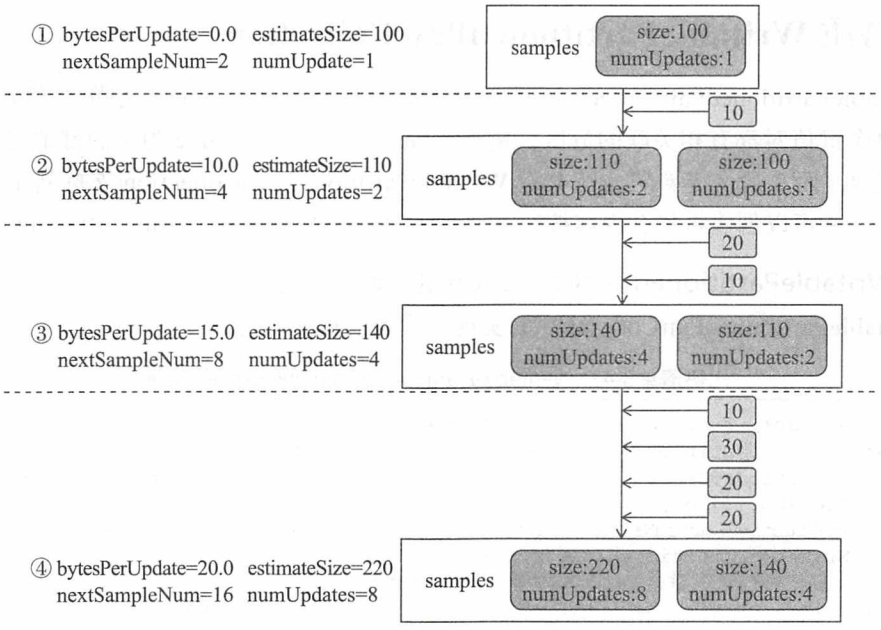


图 8-8 SizeTracker 的工作原理

根据图 8-8，对各个标记进行说明。

标记①：SizeTracker 所追踪的集合处于初始化状态，集合中没有任何元素，此时 SizeTracker 在初始化时会调用 resetSamples 方法，因而会调用 takeSample 方法进行采样。此时 samples 中只有一次采样，各个属性分别为 bytesPerUpdate=0.0；estimateSize=100；nextSampleNum=2；numUpdates=1。

标记②：向集合中添加了 10 这个元素，此时的 numUpdates 增加为 2。由于 numUpdates 与 nextSampleNum 相同，因而调用 takeSample 方法进行采样。经过计算，samples 中已经有两次采样，各个属性分别为 bytesPerUpdate=10.0；estimateSize=110；nextSampleNum=4；numUpdates=2。

标记③：向集合中添加了 20 这个元素，此时的 numUpdates 增加为 3。由于 numUpdates 不等于 nextSampleNum，所以不会进行采样。向集合中继续添加了 10 这个元素，此时的 numUpdates 增加为 4。由于 numUpdates 与 nextSampleNum 相同，因而调用 takeSample 方法进行采样。经过计算，samples 保留最新的两次采样，各个属性分别为 bytesPerUpdate=15.0；estimateSize=140；nextSampleNum=8；numUpdates=4。

标记④：向集合中添加了 10、30、20 这几个元素，此时的 numUpdates 经过三次增加后值为 7。由于 numUpdates 不等于 nextSampleNum，所以不会进行采样。向集合中继续添加了 20 这个元素，此时的 numUpdates 增加为 8。由于 numUpdates 与 nextSampleNum 相同，因而调用 takeSample 方法进行采样。经过计算，samples 保留最新的两次采样，各个属性分别为 bytesPerUpdate=20.0；estimateSize=220；nextSampleNum=16；numUpdates=8。

8.8 特质 WritablePartitionedPairCollection

WritablePartitionedPairCollection 是对由键值对构成的集合进行大小跟踪的通用接口。这里的每个键值对都有相关联的分区，例如，key 为（0，#），value 为 1 的键值对，真正的键实际是 #，而 0 则是键 # 的分区 ID。WritablePartitionedPairCollection 支持基于内存进行有效排序，并可以创建将集合内容按照字节写入磁盘的 WritablePartitionedIterator。

1. WritablePartitionedPairCollection 的实现

WritablePartitionedPairCollection 的实现如代码清单 8-47 所示。

代码清单8-47 WritablePartitionedPairCollection的实现

```
private[spark] trait WritablePartitionedPairCollection[K, V] {
  def insert(partition: Int, key: K, value: V): Unit
  def partitionedDestructiveSortedIterator(keyComparator: Option[Comparator[K]])
    : Iterator[((Int, K), V)]
  def destructiveSortedWritablePartitionedIterator(keyComparator: Option[Comparator[K]])
    : WritablePartitionedIterator = {
    // 获得对集合中的数据按照分区ID的顺序进行迭代的迭代器
    val it = partitionedDestructiveSortedIterator(keyComparator)
    new WritablePartitionedIterator { // 创建并返回WritablePartitionedIterator的匿名
                                      // 实现类的实例
```

```
        private[this] var cur = if (it.hasNext) it.next() else null
        def writeNext(writer: DiskBlockObjectWriter): Unit = {
          writer.write(cur._1._2, cur._2)
          cur = if (it.hasNext) it.next() else null
        }
        def hasNext(): Boolean = cur != null
        def nextPartition(): Int = cur._1._1
      }
    }
  }
```

根据代码清单 8-47，WritablePartitionedPairCollection 中定义了以下两个接口方法。

- insert：将键值对与相关联的分区插入到集合中。
- partitionedDestructiveSortedIterator：根据给定的对 key 进行比较的比较器，返回对集合中的数据按照分区 ID 的顺序进行迭代的迭代器。此方法需要子类实现。

此外，代码清单 8-47 展示的 destructiveSortedWritablePartitionedIterator 方法用于返回 WritablePartitionedIterator 的匿名实现类的实例，其执行步骤如下。

1）调用由子类实现的 partitionedDestructiveSortedIterator 方法，获得对集合中的数据按照分区 ID 的顺序进行迭代的迭代器。

2）创建并返回 WritablePartitionedIterator 的匿名实现类的实例。这个 WritablePartitioned Iterator 的匿名实现类的 hasNext 方法用于判断迭代器是否还有下一个元素，nextPartition 用于获取下一个元素的分区 ID，writeNext 方法则使用 DiskBlockObjectWriter（在 6.10 节有详细介绍）将键值对写入磁盘。

2. 比较器

WritablePartitionedPairCollection 的伴生对象中定义了生成两种比较器的方法，如代码清单 8-48 所示。

代码清单8-48 生成比较器

```
def partitionComparator[K]: Comparator[(Int, K)] = new Comparator[(Int, K)] {
  override def compare(a: (Int, K), b: (Int, K)): Int = {
    a._1 - b._1 // 对由partition id和key构成的两个对偶对象按照partition id进行排序
  }
}
def partitionKeyComparator[K](keyComparator: Comparator[K]): Comparator[(Int, K)] = {
  new Comparator[(Int, K)] {
    override def compare(a: (Int, K), b: (Int, K)): Int = {
      val partitionDiff = a._1 - b._1 // 对partition id和key构成的两个对偶对象按照
          partition id比较
      if (partitionDiff != 0) { // 第一级比较已经区分出了胜负
        partitionDiff // 返回比较结果
      } else { // 第一级比较没有区分出胜负
        keyComparator.compare(a._2, b._2) // 再根据指定的键比较器按照key进行第二级比较
      }
    }
  }
}
```

根据代码清单 8-48，这两个生成比较器的方法如下。

1）partitionComparator：生成对由 partition id 和 key 构成的两个对偶对象按照 partition id 进行排序的比较器。

2）partitionKeyComparator：生成对由 partition id 和 key 构成的两个对偶对象先按照 partition id 进行比较，再根据指定的键比较器按照 key 进行第二级比较的比较器。

8.9 AppendOnlyMap 的实现分析

java.util.Map 是 Java 开发人员广为使用的数据结构，我们可以使用它对常用数据进行缓存，也可以利用它进行一些算法优化。然而 java.util.Map 并不支持对 null 值的缓存操作，针对这个问题，Spark 提供了 AppendOnlyMap 来对 null 值进行缓存。AppendOnlyMap 还是在内存中对任务执行结果进行聚合运算的利器，最大可以支持 375 809 638（即 0.7×2^29）个元素。

为了便于对 AppendOnlyMap 进行分析，我们先来了解 AppendOnlyMap 包含的属性。

1）initialCapacity：初始容量值。如果未指定，默认为 64。

2）mask: 计算数据存放位置的掩码。计算 mask 的表达式为 capacity – 1。

3）capacity：data 数组的当前容量。capacity 的初始值的计算方法为取 initialCapacity 的二进制位的最高位，其余位补 0 得到新的整数（记为 highBit）。如果 highBit 与 initialCapacity 相等，则 capacity 等于 initialCapacity，否则将 highBit 左移一位后作为 capacity 的值。为了便于理解，这里列举两个例子。例一为 initialCapacity 等于 64 时，其二进制为 1000000，highBit 也为 1000000，所以 capacity 等于 64（1000000 的十进制）。例二为如果 initialCapacity 等于 72，其二进制为 1001000，highBit 为 1000000，所以 capacity 等于 128（10000000 的十进制）。

4）data：用于保存 key 和聚合值的数组。data 保存各个元素的顺序为 key0，value0，key1，value1，key2，value2……data 的初始大小为 2*capacity，data 数组的实际大小之所以是 capacity 的 2 倍，是因为 key 和聚合值各占一位。

5）LOAD_FACTOR：用于计算 data 数组容量增长的阈值的负载因子。常量 LOAD_FACTOR 固定为 0.7。

6）growThreshold：data 数组容量增长的阈值。计算 growThreshold 的表达式为 growThreshold = LOAD_FACTOR * capacity。

7）curSize：记录当前已经放入 data 的 key 与聚合值的数量。

8）haveNullValue：data 数组中是否已经有了 null 值。

9）nullValue：空值。

10）destroyed：表示 data 数组是否不再使用。

11）destructionMessage：当 destroyed 为 true 时，打印的消息内容为 "Map state is invalid

from destructive sorting!"。

在 AppendOnlyMap 的伴生对象中还有一个常量 MAXIMUM_CAPACITY，此常量的值为 1 << 29（即 2^29）。data 数组的容量不能超过 MAXIMUM_CAPACITY，以防止 data 数组溢出。

8.9.1 AppendOnlyMap 的容量增长

AppendOnlyMap 的 incrementSize 方法（见代码清单 8-49）用于扩充 AppendOnlyMap 的容量。

代码清单8-49 incrementSize方法

```
private def incrementSize() {
  curSize += 1
  if (curSize > growThreshold) {
    growTable()
  }
}
```

根据代码清单 8-49，我们知道当 curSize＞growThreshold 时，调用 growTable 方法（见代码清单 8-50）将 data 数组的容量扩大一倍，即 newCapacity = capacity * 2。

代码清单8-50 growTable方法

```
protected def growTable() {
  val newCapacity = capacity * 2
  require(newCapacity <= MAXIMUM_CAPACITY, s"Can't contain more than ${growThres-
      hold} elements")
  val newData = new Array[AnyRef](2 * newCapacity) // 创建一个两倍于当前容量
      （capacity）的新数组
  val newMask = newCapacity - 1 // 计算新数组的掩码
  var oldPos = 0
  while (oldPos < capacity) { // 将老数组中的元素拷贝到新数组的指定索引位置
    if (!data(2 * oldPos).eq(null)) {
      val key = data(2 * oldPos)
      val value = data(2 * oldPos + 1)
      var newPos = rehash(key.hashCode) & newMask
      var i = 1
      var keepGoing = true
      while (keepGoing) {
        val curKey = newData(2 * newPos)
        if (curKey.eq(null)) {
          newData(2 * newPos) = key
          newData(2 * newPos + 1) = value
          keepGoing = false
        } else {
          val delta = i
          newPos = (newPos + delta) & newMask
          i += 1
        }
      }
    }
```

```
      oldPos += 1
    }
    data = newData // 将新数组作为扩充容量后的data数组
    capacity = newCapacity // 将新数组的容量大小改为data数组的容量大小
    mask = newMask // 将掩码修改为新计算的掩码
    growThreshold = (LOAD_FACTOR * newCapacity).toInt // 重新计算AppendOnlyMap的容量
                                                      // 增长阈值
  }
```

根据代码清单 8-50，growTable 方法的执行步骤如下。

1）创建一个两倍于当前容量（capacity）的新数组，并且计算新数组的掩码。

2）将老数组中的元素拷贝到新数组的指定索引位置，此索引位置采用新的 mask 重新使用 rehash(k.hashCode) & mask 计算。在拷贝的过程中如果发生了“碰撞”，则会重新计算元素放置到新数组的索引位置，直到没有碰撞发生时将元素放入新数组。

3）将新数组作为扩充容量后的 data 数组。

4）将 data 数组的容量大小改为新数组的容量大小。

5）将掩码修改为新计算的掩码。

6）重新计算 AppendOnlyMap 的 growThreshold。

8.9.2 AppendOnlyMap 的数据更新

AppendOnlyMap 的 update 方法实现了将 key 对应的值更新到 data 数组中，其实现如代码清单 8-51 所示。

代码清单8-51 数据更新

```
def update(key: K, value: V): Unit = {
  assert(!destroyed, destructionMessage)
  val k = key.asInstanceOf[AnyRef]
  if (k.eq(null)) { // 对key是null值的更新处理
    if (!haveNullValue) {
      incrementSize()
    }
    nullValue = value
    haveNullValue = true
    return
  }
  var pos = rehash(key.hashCode) & mask // 根据key的哈希值与掩码计算元素放入data数组的索
                                        // 引位置pos
  var i = 1
  while (true) { // 将key放入data数组
    val curKey = data(2 * pos)
    if (curKey.eq(null)) {
      data(2 * pos) = k
      data(2 * pos + 1) = value.asInstanceOf[AnyRef]
      incrementSize()  // Since we added a new key
      return
    } else if (k.eq(curKey) || k.equals(curKey)) {
      data(2 * pos + 1) = value.asInstanceOf[AnyRef]
```

```
            return
          } else {
            val delta = i
            pos = (pos + delta) & mask
            i += 1
          }
        }
      }
```

根据代码清单 8-51，update 方法的执行步骤如下。

1）如果更新的 key 是 null 值，那么执行如下操作。

① 如果 data 数组中还没有 null 值（即 haveNullValue 为 false），那么调用 incrementSize 方法（见代码清单 8-49）扩充 AppendOnlyMap 的容量。

② 将 nullValue 设置为传入的 value。

③ 设置当前 data 数组中已经有了 null 值，即将 haveNullValue 置为 true。

④ 返回 nullValue。

2）根据 key 的哈希值与掩码计算元素放入 data 数组的索引位置 pos。计算表达式如下 pos = rehash(k.hashCode) & mask。

3）将 key 放入 data 数组中，具体的实现如下。

① 获取 data(2*pos) 位置的当前 key，即 curKey。

② 如果 curKey 为 null，说明 data 数组的 2*pos 的索引位置还没有放置元素，k 是首次聚合到 data 数组中，所以首先将 k 放到 data(2*pos) 位置，而将 value 放到 data(2*pos + 1) 的位置，然后调用 incrementSize 方法扩充 AppendOnlyMap 的容量后返回。

③ 如果 curKey 不等于 null 并且等于 k，说明 data 数组的 2*pos 的索引位置已经放置了元素且元素就是 k，所以将 value 更新到 data(2*pos+1) 的位置后返回。

④ 如果 curKey 不等于 null 并且不等于 k，说明 data 数组的 2*pos 的索引位置已经放置了元素，但元素不是 k，那么从 data 数组的 pos 位置向后找，直到某个位置的索引值与 mask 按位 & 运算后的新位置的 key 符合第②步或者第③步的条件。

以上虽然对 update 方法实现的数据更新进行了介绍，但为了便于理解，这里用图 8-9 来加深读者对 AppendOnlyMap 的数据更新原理的理解。

图 8-9 中的 data 数组以默认的容量 64 为例，所以 data 数组实际的大小是 128。这里对图中的标记进行说明。

标记①：调用 AppendOnlyMap 的 update 方法，传入的 key 为（0,Apache），value 为 1，计算得到 pos 为 6，由于 2×6=12 的索引位置没有元素，因此将（0,Apache）放入 data 数组索引为 12 的位置，将 1 放入索引为 2×6+1=13 的位置。

标记②：调用 AppendOnlyMap 的 update 方法，传入的 key 为（0,Apache），value 为 3，计算得到 pos 为 6，由于 2×6=12 的索引位置的元素与（0,Apache）一样，因此将 3 更新到索引为 2×6+1=13 的位置。

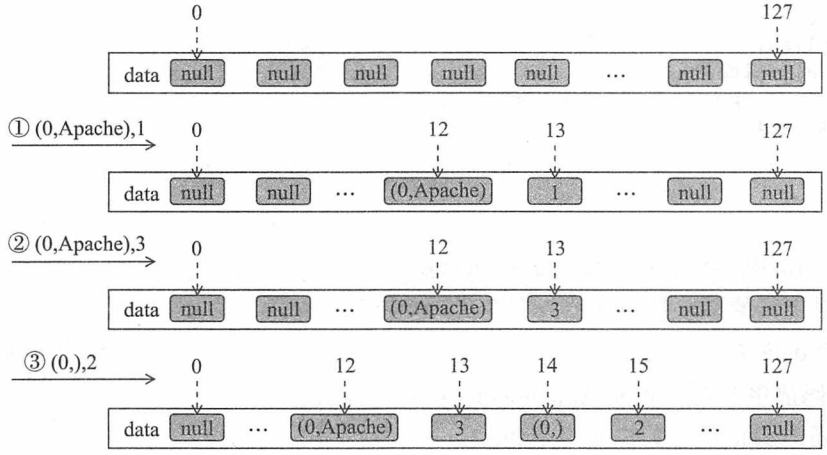


图 8-9 AppendOnlyMap 的数据更新原理

标记③：调用 AppendOnlyMap 的 update 方法，传入的 key 为（0,），value 为 2，计算得到 pos 为 6，由于 2×6＝12 的索引位置已经放入了（0,Apache），因此向后寻找新的位置。pos 为 7 时，由于 2×7＝14 的索引位置没有元素，因此将（0,）放入 data 数组索引为 14 的位置，将 2 放入索引为 2×7＋1＝15 的位置。

8.9.3 AppendOnlyMap 的缓存聚合算法

AppendOnlyMap 的 changeValue 方法实现了缓存聚合算法，其实现如代码清单 8-52 所示。

代码清单8-52 数据聚合

```
def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {
  assert(!destroyed, destructionMessage)
  val k = key.asInstanceOf[AnyRef]
  if (k.eq(null)) { // 对key是null值的缓存集合
    if (!haveNullValue) {
      incrementSize()
    }
    nullValue = updateFunc(haveNullValue, nullValue)
    haveNullValue = true
    return nullValue
  }
  // 根据key的哈希值与掩码计算元素放入data数组的索引位置pos
  var pos = rehash(k.hashCode) & mask
  var i = 1
  while (true) { // 将key放入data数组中，并进行聚合
    val curKey = data(2 * pos)
    if (curKey.eq(null)) {
      val newValue = updateFunc(false, null.asInstanceOf[V])
      data(2 * pos) = k
      data(2 * pos + 1) = newValue.asInstanceOf[AnyRef]
      incrementSize()
      return newValue
```

```
    } else if (k.eq(curKey) || k.equals(curKey)) {
      val newValue = updateFunc(true, data(2 * pos + 1).asInstanceOf[V])
      data(2 * pos + 1) = newValue.asInstanceOf[AnyRef]
      return newValue
    } else {
      val delta = i
      pos = (pos + delta) & mask
      i += 1
    }
  }
  null.asInstanceOf[V] // Never reached but needed to keep compiler happy
}
```

根据代码清单 8-52，changeValue 方法接收以下两个参数。

1）key：待聚合的 key。

2）updateFunc：聚合函数。updateFunc 接收两个参数，分别是 Boolean 类型和泛型类型 V。Boolean 类型的参数表示 key 是否已经添加到 AppendOnlyMap 的 data 数组中进行过聚合。V 则表示 key 曾经添加到 AppendOnlyMap 的 data 数组进行聚合时生成的聚合值，新一轮的聚合将在之前的聚合值上累积。

有了对这两个参数的理解，就可以对 changeValue 方法的执行步骤进行分析，其执行步骤如下。

1）如果更新的 key 是 null 值，那么执行如下操作。

① 如果 data 数组中还没有 null 值（即 haveNullValue 为 false），那么调用 incrementSize 方法（见代码清单 8-49）扩充 AppendOnlyMap 的容量。

② 调用 updateFunc 函数对 nullValue 进行聚合。haveNullValue 属性作为 updateFunc 函数的 Boolean 类型参数。

③ 设置当前 data 数组中已经有了 null 值，即将 haveNullValue 置为 true。

④ 返回 nullValue。

2）根据 key 的哈希值与掩码计算元素放入 data 数组的索引位置 pos。计算表达式为 pos = rehash(k.hashCode) & mask。

3）将 key 放入 data 数组中，具体的实现如下。

① 获取 data(2*pos) 位置的当前 key，即 curKey。

② 如果 curKey 为 null，说明 data 数组的 2*pos 的索引位置还没有放置元素，k 是首次聚合到 data 数组中，所以首先调用 updateFunc 函数时指定的 Boolean 类型参数值为 false 且没有曾经的聚合值（即 V 是 null），然后将 k 放到 data(2*pos) 位置，而将调用 updateFunc 函数获得的聚合值 newValue 放到 data(2*pos+1) 的位置，最后调用 incrementSize 方法扩充 AppendOnlyMap 的容量后返回 newValue。

③ 如果 curKey 不等于 null 并且等于 k，说明 data 数组的 2*pos 的索引位置已经放置了元素，且元素就是 k，所以首先调用 updateFunc 函数时指定的 Boolean 类型参数值为 true，且曾经的聚合值就是 data(2*pos+1) 的元素，然后将调用 updateFunc 函数获得的聚合值

newValue 更新到 data(2*pos+1) 的位置，最后返回 newValue。

④ 如果 curKey 不等于 null 并且不等于 k，说明 data 数组的 2*pos 的索引位置已经放置了元素，但元素不是 k，那么从 data 数组的 pos 位置向后找，直到某个位置的索引值与 mask 按位 & 运算后的新位置的 key 符合第②步或者第③步的条件。

以上虽然对 changeValue 方法实现的缓存聚合算法进行了介绍，但还是晦涩难懂。为了便于理解，这里假设调用 changeValue 方法时传递的 updateFunc 函数只是对 key 值进行计数，那么就可以用图 8-10 来加深读者对缓存聚合算法执行原理的理解。

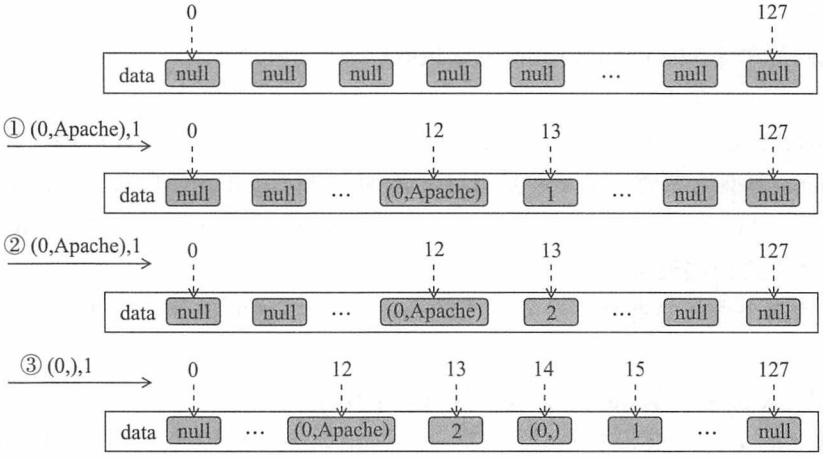


图 8-10 AppendOnlyMap 的缓存聚合算法

图 8-10 中的 data 数组以默认的容量 64 为例，所以 data 数组实际的大小是 128。这里对图中的标记进行说明。

标记①：调用 AppendOnlyMap 的 changeValue 方法，传入的 key 为（0,Apache），value 为 1。计算得到 pos 为 6，由于 2×6=12 的索引位置没有元素，因此将（0,Apache）放入 data 数组索引为 12 的位置，将计数值 1 放入索引为 2×6+1=13 的位置。

标记②：调用 AppendOnlyMap 的 changeValue 方法，传入的 key 为（0,Apache），value 为 1。计算得到 pos 为 6，由于 2×6=12 的索引位置的元素与（0,Apache）一样，因此将计数值更新为 1+1=2，并将计数值 2 更新到索引为 2×6+1=13 的位置。

标记③：调用 AppendOnlyMap 的 changeValue 方法，传入的 key 为（0,），value 为 1。计算得到 pos 为 6，由于 2×6=12 的索引位置已经放入了（0,Apache），因此向后寻找新的位置。pos 为 7 时，由于 2×7=14 的索引位置没有元素，因此将（0,）放入 data 数组索引为 14 的位置，将计数值 1 放入索引为 2×7+1=15 的位置。

8.9.4 AppendOnlyMap 的内置排序

AppendOnlyMap 的 destructiveSortedIterator 方法（见代码清单 8-53）提供了一种在不

使用额外的内存和不牺牲 AppendOnlyMap 的有效性的前提下，对 AppendOnlyMap 的 data 数组中的数据进行排序的实现。

代码清单8-53 排序

```
def destructiveSortedIterator(keyComparator: Comparator[K]): Iterator[(K, V)] = {
  destroyed = true
  var keyIndex, newIndex = 0
  while (keyIndex < capacity) { // 将data数组中的元素向前（即向着索引为0的方向）整理排列
    if (data(2 * keyIndex) != null) {
      data(2 * newIndex) = data(2 * keyIndex)
      data(2 * newIndex + 1) = data(2 * keyIndex + 1)
      newIndex += 1
    }
    keyIndex += 1
  }
  assert(curSize == newIndex + (if (haveNullValue) 1 else 0))
  // 执行比较、排序
  new Sorter(new KVArraySortDataFormat[K, AnyRef]).sort(data, 0, newIndex, key
      Comparator)
  new Iterator[(K, V)] { // 生成迭代访问data数组中的迭代器
    var i = 0
    var nullValueReady = haveNullValue
    def hasNext: Boolean = (i < newIndex || nullValueReady)
    def next(): (K, V) = {
      if (nullValueReady) {
        nullValueReady = false
        (null.asInstanceOf[K], nullValue)
      } else {
        val item = (data(2 * i).asInstanceOf[K], data(2 * i + 1).asInstanceOf[V])
        i += 1
        item
      }
    }
  }
}
```

根据代码清单 8-53，destructiveSortedIterator 方法的处理步骤如下。

1）将 data 数组中的元素向前（即向着索引为 0 的方向）整理排列。

2）利用 Sorter、KVArraySortDataFormat 及指定的比较器进行排序。这其中用到了 TimSort，也就是优化版的归并排序。

3）生成迭代访问 data 数组中的迭代器，从此迭代器访问的最后一个元素是 null 值。

8.9.5 AppendOnlyMap 的扩展

针对 AppendOnlyMap，Spark 通过继承对 AppendOnlyMap 的功能进行了扩展。

1. SizeTrackingAppendOnlyMap 的实现

AppendOnlyMap 有一个直接的子类 SizeTrackingAppendOnlyMap，看到这样的一个类名，读者会想到什么？前文介绍了 SizeTracker 和 AppendOnlyMap，想必 SizeTrackingAppend-

OnlyMap 是前两者的结合体，既可以在内存中对任务执行结果进行更新或聚合运算，也可以对自身的大小进行样本采集和大小估算。

SizeTrackingAppendOnlyMap 的实现如代码清单 8-54 所示。

代码清单8-54 SizeTrackingAppendOnlyMap的实现

```
private[spark] class SizeTrackingAppendOnlyMap[K, V]
  extends AppendOnlyMap[K, V] with SizeTracker
{
  override def update(key: K, value: V): Unit = {
    super.update(key, value)
    super.afterUpdate()
  }
  override def changeValue(key: K, updateFunc: (Boolean, V) => V): V = {
    val newValue = super.changeValue(key, updateFunc)
    super.afterUpdate()
    newValue
  }
  override protected def growTable(): Unit = {
    super.growTable()
    resetSamples()
  }
}
```

根据代码清单 8-54，SizeTrackingAppendOnlyMap 的确继承了 AppendOnlyMap 和 SizeTracker。SizeTrackingAppendOnlyMap 采用代理模式重写了 AppendOnlyMap 的三个方法（包括 update、changeValue 及 growTable）。SizeTrackingAppendOnlyMap 确保对 data 数组进行数据更新、缓存聚合等操作后，调用 SizeTracker 的 afterUpdate 方法完成采样；在扩充 data 数组的容量后，调用 SizeTracker 的 resetSamples 方法对样本进行重置，以便于对 AppendOnlyMap 的大小估算更加准确。

2. PartitionedAppendOnlyMap 的实现

Spark 对 SizeTrackingAppendOnlyMap 的功能还作了一些扩展，以便支持特质 WritablePartitionedPairCollection 的能力。PartitionedAppendOnlyMap 继承 SizeTrackingAppendOnlyMap 的同时也继承了 WritablePartitionedPairCollection，其实现如代码清单 8-55 所示。

代码清单8-55 PartitionedAppendOnlyMap的实现

```
private[spark] class PartitionedAppendOnlyMap[K, V]
  extends SizeTrackingAppendOnlyMap[(Int, K), V] with WritablePartitionedPairColle
      ction[K, V] {
  def partitionedDestructiveSortedIterator(keyComparator: Option[Comparator[K]])
    : Iterator[((Int, K), V)] = {
    val comparator = keyComparator.map(partitionKeyComparator).getOrElse(partition
        Comparator)
    destructiveSortedIterator(comparator) // 对底层的data数组进行整理和排序后获得迭代器
  }
  def insert(partition: Int, key: K, value: V): Unit = {
    update((partition, key), value)
  }
}
```

根据代码清单 8-55，PartitionedAppendOnlyMap 实现了特质 WritablePartitionedPairCollection 定义的 partitionedDestructiveSortedIterator 接口和 insert 接口。partitionedDestructiveSortedIterator 方法的执行步骤如下。

1）调用 WritablePartitionedPairCollection 的伴生对象的 partitionKeyComparator 方法（见代码清单 8-48）生成比较器。如果没有指定 key 比较器，那么调用 WritablePartitionedPairCollection 的伴生对象的 partitionComparator 方法（见代码清单 8-48）生成比较器。

2）调用 AppendOnlyMap 的 destructiveSortedIterator 方法（见代码清单 8-53）对底层的 data 数组进行整理和排序后获得迭代器。

insert 方法的实现则非常简单，它将 key 的分区 ID 和 key 作为父类 SizeTrackingAppendOnlyMap 的 update 方法的第一个参数，而将 value 作为第二个参数。以 Apache 这个 key 为例，如果通过分区计算器计算得到的分区 ID 为 0，那么（0, Apache）这个对偶将存储在底层 data 数组索引为 12 的位置，而 value 存储在 data 数组索引为 13 的位置。

8.10 PartitionedPairBuffer 的实现分析

map 任务除了采用 AppendOnlyMap 对键值对在内存中进行更新或聚合，Spark 还提供了一种将键值对缓存在内存中，并支持对元素进行排序的数据结构。AppendOnlyMap 的表现行为类似于 Map，而这种数据结构类似于 Collection，它就是 PartitionedPairBuffer。PartitionedPairBuffer 最大支持 1 073 741 823（即 2^30–1）个元素。

PartitionedPairBuffer 同时继承了 WritablePartitionedPairCollection 和 SizeTracker 这两个特质。

为了容易理解 PartitionedPairBuffer 的实现，我们先来了解这个数据结构中的属性。

- initialCapacity：初始容量值。如果未指定，默认为 64。
- capacity：data 数组的当前容量。capacity 的初始值等于 initialCapacity。
- curSize：记录当前已经放入 data 的 key 与 value 的数量。
- data：用于保存 key 和 value 的数组。data 保存各个元素的顺序为 key0，value0，key1，value1，key2，value2……data 的初始大小为 2 * capacity，data 数组的实际大小之所以是 capacity 的 2 倍，是因为 key 和 value 各占一位。

在 PartitionedPairBuffer 的伴生对象中还有一个常量 MAXIMUM_CAPACITY，此常量的值为 2^30–1。data 数组的容量不能超过 MAXIMUM_CAPACITY，以防止 data 数组溢出。

8.10.1 PartitionedPairBuffer 的容量增长

PartitionedPairBuffer 的 growArray 方法（见代码清单 8-56）用于扩充 PartitionedPairBuffer 的容量。

代码清单8-56 PartitionedPairBuffer的容量增长

```
private def growArray(): Unit = {
  if (capacity >= MAXIMUM_CAPACITY) { // 防止PartitionedPairBuffer的容量超过MAXIMUM_
      CAPACITY的限制
    throw new IllegalStateException(s"Can't insert more than ${MAXIMUM_CAPACITY} elements")
  }
  val newCapacity = // 计算对PartitionedPairBuffer进行扩充后的容量大小
    if (capacity * 2 < 0 || capacity * 2 > MAXIMUM_CAPACITY) { // Overflow
      MAXIMUM_CAPACITY
    } else {
      capacity * 2
    }
  val newArray = new Array[AnyRef](2 * newCapacity) // 创建一个两倍于新容量的大小的新数组
  // 将底层data数组的每个元素都拷贝到新数组中
  System.arraycopy(data, 0, newArray, 0, 2 * capacity)
  data = newArray // 将新数组设置为底层的data数组
  capacity = newCapacity // 将PartitionedPairBuffer的当前容量设置为新的容量大小
  resetSamples() // 对样本进行重置，以便估算准确
}
```

根据代码清单 8-56，growArray 方法的执行步骤如下。

1）对 PartitionedPairBuffer 的当前容量（即 capacity）进行校验，以防止超过常量 MAXIMUM_CAPACITY 的限制。

2）计算对 PartitionedPairBuffer 进行扩充后的容量 newCapacity，即 capacity * 2。如果 newCapacity 超过了 MAXIMUM_CAPACITY 的限制，那么将 newCapacity 重新设置为 MAXIMUM_CAPACITY。

3）创建一个两倍于 newCapacity 的大小的新数组。

4）将底层 data 数组的每个元素都拷贝到新数组中。

5）将新数组设置为底层的 data 数组，并将当前容量 capacity 设置为 newCapacity。

6）调用父类 SizeTracker 的 resetSamples 对样本进行重置，以便估算准确。

8.10.2 PartitionedPairBuffer 的插入

PartitionedPairBuffer 的 insert 方法（见代码清单 8-57）用于将 key 的分区 ID、key 及 value 添加到 PartitionedPairBuffer 底层的 data 数组中。

代码清单8-57 PartitionedPairBuffer的insert方法

```
def insert(partition: Int, key: K, value: V): Unit = {
  if (curSize == capacity) { // 如果底层data数组已经满了，对PartitionedPairBuffer的容
                             // 量进行扩充
    growArray()
  }
  data(2 * curSize) = (partition, key.asInstanceOf[AnyRef]) // 将key及其分区ID作为
                                                            // 对偶放入data数组
  data(2 * curSize + 1) = value.asInstanceOf[AnyRef] // 将value放入data数组
  curSize += 1 // 增加已经放入data数组的key与value的数量
  afterUpdate() // 对集合大小进行采样
}
```

根据代码清单 8-57，insert 方法的执行步骤如下。

1）如果底层 data 数组已经满了，即 curSize 与 capacity 相等时，调用 growArray 方法（见代码清单 8-56）对 PartitionedPairBuffer 的容量进行扩充。

2）将 key 的分区 ID 和 key 作为对偶放入 data 数组的索引为 2 * curSize 的位置。

3）将 value 放入 data 数组的索引为 2 * curSize + 1 的位置。

4）增加已经放入 data 数组的 key 与 value 的数量，即将 curSize 加一。

5）调用父类 SizeTracker 的 afterUpdate 对集合大小进行采样。

可以用图 8-11 来表示数据插入 PartitionedPairBuffer 的过程。

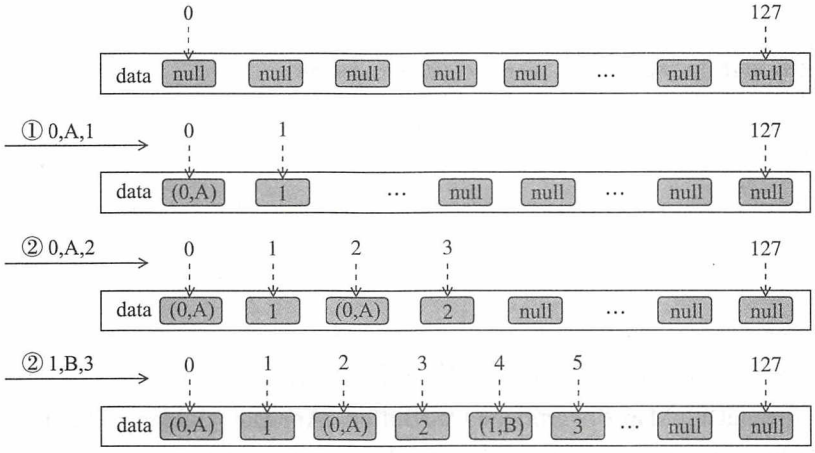


图 8-11 PartitionedPairBuffer 的数据插入

8.10.3 PartitionedPairBuffer 的迭代器

PartitionedPairBuffer 实现了父类 WritablePartitionedPairCollection 定义的 partitionedDestructiveSortedIterator 接口，根据给定的对 key 进行比较的比较器，返回对集合中的数据按照分区 ID 的顺序进行迭代的迭代器。

PartitionedPairBuffer 实现的 partitionedDestructiveSortedIterator 方法，如代码清单 8-58 所示。

代码清单8-58 partitionedDestructiveSortedIterator的实现

```
override def partitionedDestructiveSortedIterator(keyComparator: Option[Comparator[K]])
  : Iterator[((Int, K), V)] = {
  val comparator = keyComparator.map(partitionKeyComparator).getOrElse(partition
      Comparator)
  new Sorter(new KVArraySortDataFormat[(Int, K), AnyRef]).sort(data, 0, curSize,
      comparator)
  iterator
}
```

根据代码清单 8-58，partitionedDestructiveSortedIterator 方法的执行步骤为如下。

仅供非商业用途或交流学习使用

1）调用 WritablePartitionedPairCollection 的伴生对象的 partitionKeyComparator 方法（见代码清单 8-48）生成比较器。如果没有指定 key 比较器，那么调用 WritablePartitionedPairCollection 的伴生对象的 partitionComparator 方法（见代码清单 8-48）生成比较器。

2）利用 Sorter、KVArraySortDataFormat 及第 1) 步生成的比较器执行内置排序。这其中用到了 TimSort，也就是优化版的归并排序。

3）调用 PartitionedPairBuffer 的 iterator 方法（见代码清单 8-59）获得对 data 中的数据进行迭代的迭代器。

代码清单8-59 PartitionedPairBuffer的迭代器

```
private def iterator(): Iterator[((Int, K), V)] = new Iterator[((Int, K), V)] {
  var pos = 0
  override def hasNext: Boolean = pos < curSize
  override def next(): ((Int, K), V) = {
    if (!hasNext) {
      throw new NoSuchElementException
    }
    val pair = (data(2 * pos).asInstanceOf[(Int, K)], data(2 * pos + 1).asInstanceOf[V])
    pos += 1
    pair
  }
}
```



我们已经介绍了在将 map 任务的输出数据写入磁盘前，将数据临时存放的在内存中的两种数据结构 AppendOnlyMap 和 PartitionedPairBuffer。AppendOnlyMap 和 PartitionedPairBuffer 底层都使用数组存放元素，两者都有相似的容量增长实现，都有生成访问底层 data 数组的迭代器方法，那么这两者有哪些区别呢？根据之前的分析，它们的主要区别如下。

1）AppendOnlyMap 会对元素在内存中进行更新或聚合，而 PartitionedPairBuffer 只起到数据缓冲的作用。

2）AppendOnlyMap 的行为更像 map，元素以散列的方式放入到 data 数组，而 PartitionedPairBuffer 的行为更像 collection，元素都是从 data 数组的起始索引 0 和 1 开始连续放入的。

3）AppendOnlyMap 没有继承 SizeTracker，因而不支持采样和大小估算，而 PartitionedPairBuffer 天生就继承自 SizeTracker，所以支持采样和大小估算。好在 AppendOnly-Map 继承了 SizeTracker 的子类 SizeTrackingAppendOnlyMap。

4）AppendOnlyMap 没有继承 WritablePartitionedPairCollection，因而不支持基于内存进行有效排序的迭代器，也不可以创建将集合内容按照字节写入磁盘的 WritablePartitionedIterator。而 PartitionedPairBuffer 天生就继承自 WritablePartitionedPairCollection。好在 AppendOnlyMap 继承了 WritablePartitionedPairCollection 的子类 PartitionedAppendOnlyMap。

8.11 外部排序器

Spark 中的外部排序器用于对 map 任务的输出数据在 map 端或 reduce 端进行排序。Spark

仅供非商业用途或交流学习使用

中有两个外部排序器，分别是 ExternalSorter 和 ShuffleExternalSorter，本节将分别展开介绍。

8.11.1 ExternalSorter 详解

ExternalSorter 是 SortShuffleManager 的底层组件，它提供了很多功能，包括将 map 任务的输出存储到 JVM 的堆中，如果指定了聚合函数，则还会对数据进行聚合；使用分区计算器首先将 Key 分组到各个分区中，然后使用自定义比较器对每个分区中的键进行可选的排序；可以将每个分区输出到单个文件的不同字节范围中，便于 reduce 端的 Shuffle 获取。

ExternalSorter 中有很多属性，分别如下。

- context：即任务上下文（TaskContext），由于 TaskContext 只有 TaskContextImpl 这一个实现类，因此我们也可以认为是 TaskContextImpl。
- aggregator：对 map 任务的输出数据进行聚合的聚合器，类型为 Aggregator[K, V, C]。
- partitioner：对 map 任务的输出数据按照 key 计算分区的分区计算器 Partitioner。
- ordering：对 map 任务的输出数据按照 key 进行排序的 scala.math.Ordering 的实现类。
- serializer：即 SparkEnv 的子组件 serializer。
- serInstance：serializer 的实例。
- conf：即 SparkConf。
- numPartitions：分区数量。实际通过调用 partitioner 的 numPartitions 方法（请参阅 7.2.4 节）获得，默认为 1。
- shouldPartition：是否有分区。当 numPartitions 大于 1 时为 true。
- blockManager：即 SparkEnv 的子组件 BlockManager。
- diskBlockManager：即 BlockManager 的子组件 DiskBlockManager。
- serializerManager：即 SparkEnv 的子组件 SerializerManager。
- fileBufferSize：用于设置 DiskBlockObjectWriter 内部的文件缓冲大小。可通过 spark.shuffle.file.buffer 属性进行配置，默认是 32KB。
- serializerBatchSize：用于将 DiskBlockObjectWriter 内部的文件缓冲写到磁盘的大小。可通过 spark.shuffle.spill.batchSize 属性进行配置，默认是 10000。
- map：类型为 PartitionedAppendOnlyMap[K, C]。当设置了聚合器（Aggregator）时，map 端将中间结果溢出到磁盘前，先利用此数据结构在内存中对中间结果进行聚合处理。
- buffer：类型为 PartitionedPairBuffer[K, C]。当没有设置聚合器（Aggregator）时，map 端将中间结果溢出到磁盘前，先利用此数据结构将中间结果存储在内存中。
- _diskBytesSpilled：用于对溢出到磁盘的字节数进行统计（单位为字节）。diskBytesSpilled 方法专门用于返回 _diskBytesSpilled 的值。
- _peakMemoryUsedBytes：内存中数据结构大小的峰值（单位为字节）。peakMemoryUsedBytes 方法专门用于返回 _peakMemoryUsedBytes 的值。
- isShuffleSort：是否对 Shuffle 数据进行排序。



仅供非商业用途或交流学习使用

- forceSpillFiles：缓存强制溢出的文件数组。forceSpillFiles 的类型为 ArrayBuffer[SpilledFile]。SpilledFile 保存了溢出文件的信息，包括 file（文件）、blockId（BlockId）、serializerBatchSizes、elementsPerPartition（每个分区的元素数量）。
- readingIterator：类型为 SpillableIterator，用于包装内存中数据的迭代器和溢出文件，并表现为一个新的迭代器。
- keyComparator：中间输出的 key 的比较器。keyComparator 的类型为 Comparator[K]，用于在分区内对中间结果按照 key 进行排序，以便于聚合。keyComparator 的定义如代码清单 8-60 所示。

代码清单8-60 keyComparator的定义

```
private val keyComparator: Comparator[K] = ordering.getOrElse(new Comparator[K] {
    override def compare(a: K, b: K): Int = {
      val h1 = if (a == null) 0 else a.hashCode()
      val h2 = if (b == null) 0 else b.hashCode()
      if (h1 < h2) -1 else if (h1 == h2) 0 else 1
    }
  })
```

根据 keyComparator 的定义，当用户没有指定 ordering 时，将会创建一个按照 Key 的哈希值进行比较的默认比较器。因为不同的 key 值也可能有相同的哈希值，因此默认的比较器会工作不正常。此比较器经常作为代码清单 8-48 中介绍的 partitionComparator 和 partitionKeyComparator 所需要的 key 值比较器。



- spills：缓存溢出的文件数组。spills 的类型为 ArrayBuffer[SpilledFile]。numSpills 方法用于返回 spills 的大小，即溢出的文件数量。

由于 ExternalSorter 继承了抽象类 Spillable，所以对 Spillable 中的属性也需要理解。Spillable 中的属性如下所示。

- initialMemoryThreshold：对集合的内存使用进行跟踪的初始内存阈值。可通过 spark.shuffle.spill.initialMemoryThreshold 属性配置，默认为 5MB。
- numElementsForceSpillThreshold：当集合中有太多元素时，强制将集合中的数据溢出到磁盘的阈值。可通过 spark.shuffle.spill.numElementsForceSpillThreshold 属性配置，默认为 Long.MAX_VALUE。
- myMemoryThreshold：对集合的内存使用进行跟踪的初始内存阈值。myMemoryThreshold 的初始值等于 initialMemoryThreshold。
- _elementsRead：已经插入到集合的元素数量。addElementsRead 方法用于将 _elementsRead 加一。elementsRead 方法用于读取 _elementsRead 的值。
- _memoryBytesSpilled：内存中的数据已经溢出到磁盘的字节总数。
- _spillCount：集合产生溢出的次数。

仅供非商业用途或交流学习使用

1. map 端输出的缓存处理

map 任务在执行结束后会将数据写入磁盘，等待 reduce 任务获取。但在写入磁盘之前，Spark 可能会对 map 任务的输出在内存中进行一些排序和聚合。ExternalSorter 的 insertAll 方法是这一过程的入口，其实现如代码清单 8-61 所示。

代码清单8-61 map端输出的缓存处理

```
def insertAll(records: Iterator[Product2[K, V]]): Unit = {
  val shouldCombine = aggregator.isDefined
  if (shouldCombine) { // 如果用户指定了聚合器，那么对数据进行聚合
    val mergeValue = aggregator.get.mergeValue
    val createCombiner = aggregator.get.createCombiner
    var kv: Product2[K, V] = null
    val update = (hadValue: Boolean, oldValue: C) => {
      if (hadValue) mergeValue(oldValue, kv._2) else createCombiner(kv._2)
    }
    while (records.hasNext) {
      addElementsRead()
      kv = records.next()
      map.changeValue((getPartition(kv._1), kv._1), update)
      maybeSpillCollection(usingMap = true)
    }
  } else { // 如果用户没有指定聚合器，只对数据进行缓冲
    while (records.hasNext) {
      addElementsRead()
      val kv = records.next()
      buffer.insert(getPartition(kv._1), kv._1, kv._2.asInstanceOf[C])
      maybeSpillCollection(usingMap = false)
    }
  }
}
```



根据代码清单 8-61，insertAll 方法的执行步骤如下。

1）如果用户指定了聚合器，则执行如下操作。

① 获取聚合器的 mergeValue 函数（此函数用于将新的 Value 合并到聚合的结果中）。

② 获取聚合器的 createCombiner 函数（此函数用于创建聚合的初始值）。

③ 定义偏函数 update。此函数的作用是当有新的 Value 时（即 hadValue 为 true），调用 mergeValue 函数将新的 Value 合并到之前聚合的结果中，否则说明刚刚开始聚合，此时调用 createCombiner 函数以 Value 作为聚合的初始值。

④ 迭代输入的记录，首先调用父类 Spillable 的 addElementsRead 方法增加已经读取的元素数，然后对每个 scala.Product2[K,V] 的 key 通过调用 getPartition 方法（见代码清单 8-62）计算分区索引（ID），并将分区索引与 key 作为调用 AppendOnlyMap 的 changeValue 方法的参数 key，以偏函数 update 作为 changeValue 方法的参数 updateFunc，对由分区索引与 key 组成的对偶进行聚合。最后调用 maybeSpillCollection 方法（将在下一节介绍）进行可能的磁盘溢出。

2）如果用户没有指定聚合器，则对迭代器中的记录进行迭代，并在每次迭代过程中执

仅供非商业用途或交流学习使用

行如下操作。

① 调用父类 Spillable 的 addElementsRead 方法增加已经读取的元素数。

② 对每个 scala.Product2[K,V] 的 key 通过调用 getPartition 方法（见代码清单 8-62）计算分区索引（ID），并将分区索引、key 及 value 作为调用 PartitionedPairBuffer 的 insert 方法的参数。

③ 调用 maybeSpillCollection 方法（将在下一节介绍）进行可能的磁盘溢出。

代码清单8-62 ExternalSorter的getPartition方法

```
private def getPartition(key: K): Int = {
  if (shouldPartition) partitioner.get.getPartition(key) else 0
}
```

2. 缓存溢出

ExternalSorter 的 map 属性的类型为 PartitionedAppendOnlyMap[K,C]，buffer 属性的类型为 PartitionedPairBuffer[K,C]。既然 ExternalSorter 使用了 AppendOnlyMap 和 PartitionedPairBuffer，并且 AppendOnlyMap 和 PartitionedPairBuffer 的容量都可以增长，那么数据量不大的时候不会有问题。由于大数据处理的数据量往往都很大，全部都放入内存将很容易引起系统的 OOM 问题。另一方面，map 任务的输出需要写入磁盘，磁盘写入频率过高会因为大量的磁盘 I/O 降低效率，那么何时才应该将内存中的数据写入到磁盘呢？ Spark 为了解决这两个问题，提供了 maybeSpillCollection 方法（见代码清单 8-63）。以判断何时将内存中的数据写入磁盘。



代码清单8-63 缓存溢出

```
private def maybeSpillCollection(usingMap: Boolean): Unit = {
  var estimatedSize = 0L
  if (usingMap) { // 如果使用了PartitionedAppendOnlyMap
    estimatedSize = map.estimateSize() // 对PartitionedAppendOnlyMap的大小进行估算
    if (maybeSpill(map, estimatedSize)) { // 将PartitionedAppendOnlyMap中的数据溢出到磁盘
      map = new PartitionedAppendOnlyMap[K, C] // 重新创建PartitionedAppendOnlyMap
    }
  } else { // 如果使用了PartitionedPairBuffer
    estimatedSize = buffer.estimateSize() // 对PartitionedPairBuffer的大小进行估算
    if (maybeSpill(buffer, estimatedSize)) { // 将PartitionedPairBuffer中的数据溢出到磁盘
      buffer = new PartitionedPairBuffer[K, C] // 重新创建PartitionedPairBuffer
    }
  }
  if (estimatedSize > _peakMemoryUsedBytes) { // 更新ExternalSorter已经使用内存大小的峰值
    _peakMemoryUsedBytes = estimatedSize
  }
}
```

根据代码清单 8-63，maybeSpillCollection 方法的执行步骤如下。

① 如果 ExternalSorter 正在使用的数据结构是 PartitionedAppendOnlyMap，那么对 PartitionedAppendOnlyMap 的大小进行估算。调用 maybeSpill 方法（见代码清单 8-64）

仅供非商业用途或交流学习使用

时，如果的确将 PartitionedAppendOnlyMap 中的数据溢出到了磁盘上，那么重新创建 PartitionedAppendOnlyMap。

② 如果 ExternalSorter 正在使用的数据结构是 PartitionedPairBuffer，那么对 PartitionedPairBuffer 的大小进行估算。调用 maybeSpill 方法（见代码清单 8-64）时，如果的确将 PartitionedPairBuffer 中的数据溢出到了磁盘上，那么重新创建 PartitionedPairBuffer。

③ 如果估算的大小超过了 ExternalSorter 已经使用的内存大小的峰值 _peakMemoryUsedBytes，那么将 ExternalSorter 的 _peakMemoryUsedBytes 修改为估算的大小。

maybeSpill 方法用于将 PartitionedAppendOnlyMap 或 PartitionedPairBuffer 底层的数据溢出到磁盘，其实现如代码清单 8-64 所示。

代码清单8-64 maybeSpill方法的实现

```
protected def maybeSpill(collection: C, currentMemory: Long): Boolean = {
  var shouldSpill = false
  if (elementsRead % 32 == 0 && currentMemory >= myMemoryThreshold) {
    val amountToRequest = 2 * currentMemory - myMemoryThreshold
    //为当前任务尝试获取期望大小的内存，得到实际获得的大小
    val granted = acquireMemory(amountToRequest)
    myMemoryThreshold += granted // 更新已经获得的内存大小
    shouldSpill = currentMemory >= myMemoryThreshold
  }
  shouldSpill = shouldSpill || _elementsRead > numElementsForceSpillThreshold
  if (shouldSpill) { // 如果应该进行溢出
    _spillCount += 1
    logSpillage(currentMemory)
    spill(collection) // 将集合中的数据溢出到磁盘
    _elementsRead = 0 // 已读取元素数（_elementsRead）归零
    _memoryBytesSpilled += currentMemory 更新已经溢出的内存大小
    releaseMemory() // 释放ExternalSorter占用的内存
  }
  shouldSpill // 返回是否进行了溢出
}
```

根据代码清单 8-64，maybeSpill 方法的执行步骤如下。

1）如果当前集合已经读取的元素数是 32 的倍数，并且集合当前的内存大小（currentMemory）大于等于 myMemoryThreshold，那么执行以下操作。

①调用 acquireMemory 方法（ExternalSorter 继承了 MemoryConsumer，所以这里的 acquire-Memory 方法见代码清单 8-30），为当前任务尝试获取 2 * currentMemory-myMemoryThreshold 大小的内存，并获得实际获得的内存大小 granted。

② 将 granted 累加到 myMemoryThreshold。

③ 判断是否应该进行溢出，即 currentMemory 是否大于等于 myMemoryThreshold。如果 currentMemory 还是大于等于 myMemoryThreshold，说明 TaskMemoryManager 已经没有多余的内存可以分配了，此时应该进行溢出。

2）如果需要溢出或者 _elementsRead 大于 numElementsForceSpillThreshold，那就需要

仅供非商业用途或交流学习使用

溢出。溢出执行的操作如下。

① 将 _spillCount 加一。

② 调用 spill 方法（见代码清单 8-65）将集合中的数据溢出到磁盘。

③ 溢出后处理，包括已读取元素数（_elementsRead）归零，已溢出内存字节数（_memoryBytesSpilled），增加当前集合的大小（currentMemory），释放 ExternalSorter 占用的内存。

3）返回是否进行了溢出。

ExternalSorter 实现了父类 Spillable 定义的 spill 接口，其实现如代码清单 8-65 所示。

代码清单8-65 溢出方法spill

```
override protected[this] def spill(collection: WritablePartitionedPairCollection[K,
    C]): Unit = {
  val inMemoryIterator = collection.destructiveSortedWritablePartitionedIterator
      (comparator)
  val spillFile = spillMemoryIteratorToDisk(inMemoryIterator) // 将集合中的数据溢出到
      磁盘
  spills += spillFile // 将溢出生成的文件添加到spills中
}
```

根据代码清单 8-65，spill 方法的参数为 WritablePartitionedPairCollection。由于 PartitionedAppendOnlyMap 和 PartitionedPairBuffer 都实现了特质 WritablePartitionedPairCollection，因此 spill 方法对这两种数据结构都适用。spill 方法的执行步骤如下。

① 调用 WritablePartitionedPairCollection 的 destructiveSortedWritablePartitionedIterator 方法（见代码清单 8-47）获取 WritablePartitionedIterator。comparator 方法（见代码清单 8-66）用于获取 key 的比较器，如果定义了 ordering 或 aggregator，那么比较器就是 keyComparator，否则没有比较器。

② 调用 spillMemoryIteratorToDisk 方法（见代码清单 8-67），通过 WritablePartitionedIterator 将集合中的数据溢出到磁盘。

③ 将溢出生成的文件添加到 spills 中。

代码清单8-66 comparator方法的定义

```
private def comparator: Option[Comparator[K]] = {
  if (ordering.isDefined || aggregator.isDefined) {
    Some(keyComparator)
  } else {
    None
  }
}
```

代码清单8-67 将内存数据溢出到磁盘

```
  private[this] def spillMemoryIteratorToDisk(inMemoryIterator: WritablePartitioned
      Iterator)
      : SpilledFile = {
    val (blockId, file) = diskBlockManager.createTempShuffleBlock() // 创建唯一的
        BlockId和文件
```

仅供非商业用途或交流学习使用


```
var objectsWritten: Long = 0 // objectsWritten用于统计已经写入磁盘的键值对数量
val spillMetrics: ShuffleWriteMetrics = new ShuffleWriteMetrics
val writer: DiskBlockObjectWriter = // 获取DiskBlockObjectWriter
  blockManager.getDiskWriter(blockId, file, serInstance, fileBufferSize,
    spillMetrics)
val batchSizes = new ArrayBuffer[Long] // 创建存储批次大小的数组缓冲batchSizes
val elementsPerPartition = new Array[Long](numPartitions) // 创建存储每个分区有
    多少个元素的数组缓冲
def flush(): Unit = {
  val segment = writer.commitAndGet()
  batchSizes += segment.length
  _diskBytesSpilled += segment.length
  objectsWritten = 0
}

var success = false
try {
  while (inMemoryIterator.hasNext) {
    val partitionId = inMemoryIterator.nextPartition() // 获取数据的分区ID
    require(partitionId >= 0 && partitionId < numPartitions,
      s"partition Id: ${partitionId} should be in the range [0, ${num
          Partitions}]")
    inMemoryIterator.writeNext(writer) // 将键值对写入磁盘
    elementsPerPartition(partitionId) += 1 // 将elementsPerPartition中统计的分
        区对应的元素数量加1
    objectsWritten += 1 // 将objectsWritten加一

    if (objectsWritten == serializerBatchSize) {
      flush() // 将DiskBlockObjectWriter的输出流中的数据真正写入到磁盘
    }
  }
  if (objectsWritten > 0) {
    flush() // 将DiskBlockObjectWriter的输出流中的数据真正写入到磁盘
  } else {
    writer.revertPartialWritesAndClose()
  }
  success = true
} finally {
  if (success) {
    writer.close()
  } else {
    writer.revertPartialWritesAndClose()
    if (file.exists()) {
      if (!file.delete()) {
        logWarning(s"Error deleting ${file}")
      }
    }
  }
}
SpilledFile(file, blockId, batchSizes.toArray, elementsPerPartition) // 创建
    并返回SpilledFile
}
```

根据代码清单 8-67，spillMemoryIteratorToDisk 方法的执行步骤如下。

1）进行一些准备工作，如下所示。

仅供非商业用途或交流学习使用

① 调用 DiskBlockManager 的 createTempShuffleBlock 方法（见代码清单 6-26）创建唯一的 BlockId 和文件。

② 声明变量 objectsWritten。objectsWritten 用于统计已经写入磁盘的键值对数量。

③ 创建 ShuffleWriteMetrics。ShuffleWriteMetrics 用于对 Shuffle 中间结果写入到磁盘的度量与统计。

④ 调用 BlockManager 的 getDiskWriter 方法（见代码清单 6-83）获取 DiskBlockObject Writer。

⑤ 创建存储批次大小的数组缓冲 batchSizes。

⑥ 创建用于存储每个分区有多少个元素的数组缓冲 elementsPerPartition。

2）对 WritablePartitionedIterator 进行迭代，每次迭代都执行以下操作。

① 调用 WritablePartitionedIterator 的 nextPartition 方法（见代码清单 8-47）获取数据的分区 ID。

② 以 DiskBlockObjectWriter 为参数，调用 WritablePartitionedIterator 的 writeNext 方法（见代码清单 8-47）将键值对写入磁盘。

③ 将 elementsPerPartition 中统计的分区对应的元素数量加 1。

④ 将 objectsWritten 加 1。

⑤ 如果 objectsWritten 与 serializerBatchSize 相等，则调用 flush 方法首先将 DiskBlockObjectWriter 的输出流中的数据真正写入到磁盘，然后将本批次写入的文件长度添加到 batchSizes 和 _diskBytesSpilled 中，最后将 objectsWritten 清零，以便下一批次的正确执行。



3）如果 objectsWritten 大于 0，则调用 flush 方法，否则调用 DiskBlockObjectWriter 的 revertPartialWritesAndClose 方法，最后将 success 设置为 true。

4）创建并返回 SpilledFile。

为了让读者对缓存溢出的原理有更深入的理解，笔者将 spark.shuffle.spill. numElements ForceSpillThreshold 设置为 5。我们首先假设给 ExternalSorter 指定了聚合函数，那么 External Sorter 将选择 PartitionedAppendOnlyMap 作为缓存，并且 Partitioned AppendOnlyMap 在溢出前，底层的 data 数组里已经散列存储了 5 个元素。那么执行缓存溢出的过程可以用图 8-12 表示。

图 8-12 表示 PartitionedAppendOnlyMap 底层 data 数组中的元素都是散列存储，执行缓存溢出的步骤如下。

1）将 data 数组中的元素向低索引端整理。

2）将 data 数组中的元素根据指定的比较器对元素进行排序。如果指定了聚合函数或者排序函数，那么排序先按照分区 ID 进行排序，然后按照 key 进行排序。

3）将 data 数组中的数据通过迭代器写到磁盘文件。图中的磁盘文件的内容是从逻辑的角度描绘的，实际上并非是明文输出的。

对于 PartitionedPairBuffer 而言，其底层 data 数组中的元素是在插入的时候就排列整

齐的，因此溢出过程中不会有整理操作，其余步骤与 PartitionedAppendOnlyMap 类似，如图 8-13 所示。

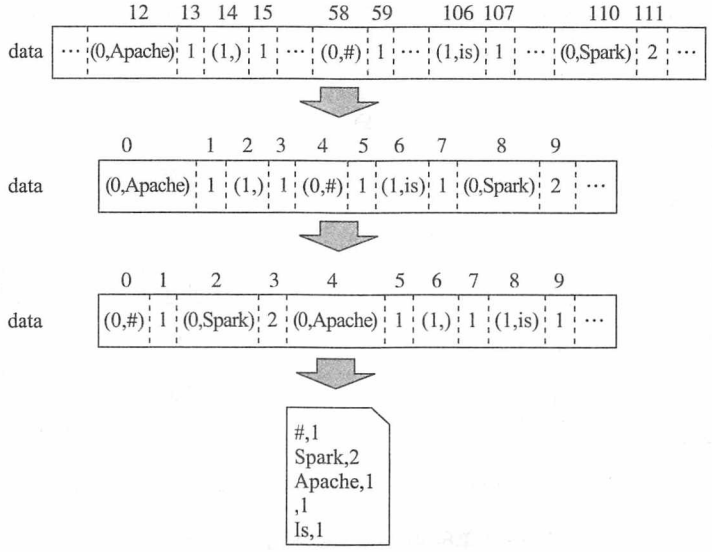


图 8-12 PartitionedAppendOnlyMap 的缓存溢出

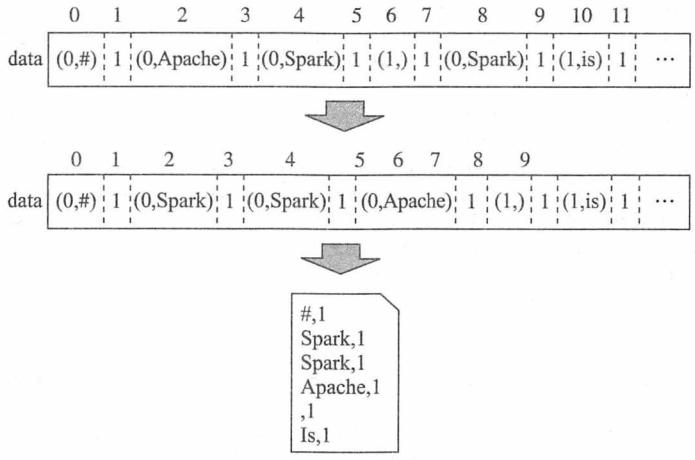


图 8-13 PartitionedPairBuffer 的缓存溢出

经过对 map 端输出的缓存处理及缓存溢出的分析，可以用图 8-14 来表示 ExternalSorter 实现的 map 端输出的整个过程。

3. map 端输出的持久化

数据仅仅存放在内存中，必然存在着丢失的风险。ExternalSorter 的 writePartitionedFile 方法（见代码清单 8-68）用于持久化计算结果。

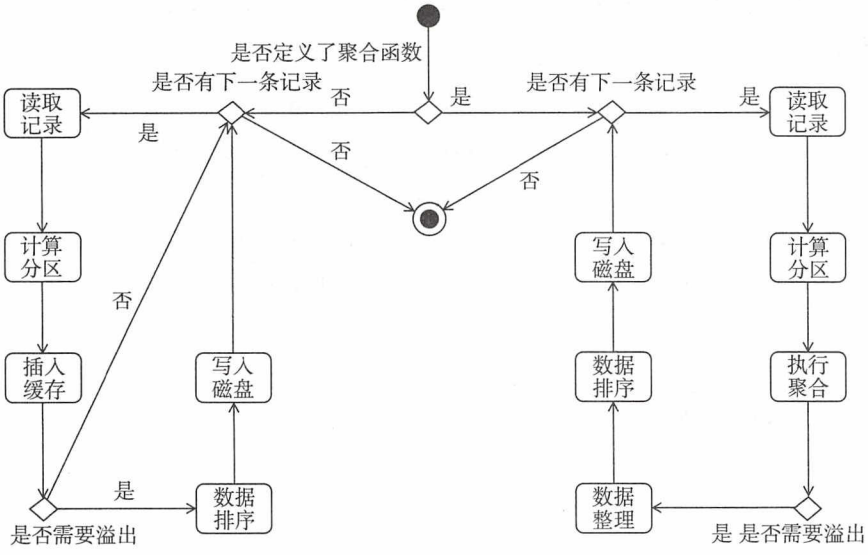


图 8-14 ExternalSorter 实现的 map 端输出的整个过程

代码清单8-68 map端输出的持久化

```
def writePartitionedFile(
    blockId: BlockId,
    outputFile: File): Array[Long] = {
  val lengths = new Array[Long](numPartitions) // 创建对每个分区的长度进行跟踪的数组lengths
  val writer = blockManager.getDiskWriter(blockId, outputFile, serInstance, file
      BufferSize,
    context.taskMetrics().shuffleWriteMetrics) // 获取DiskBlockObjectWriter
  if (spills.isEmpty) { // 没有缓存溢出到磁盘的文件，即所有的数据依然都在内存中
    val collection = if (aggregator.isDefined) map else buffer
    val it = collection.destructiveSortedWritablePartitionedIterator(comparator)
    while (it.hasNext) { // 将底层data数组中的数据按照分区ID分别写入到磁盘中
      val partitionId = it.nextPartition()
      while (it.hasNext && it.nextPartition() == partitionId) {
        it.writeNext(writer)
      }
      val segment = writer.commitAndGet()
      lengths(partitionId) = segment.length // 将分区的数据长度更新到lengths数组中
    }
  } else { // 如果spills中缓存了溢出到磁盘的文件，即有些数据在内存中，有些数据已经溢出到了磁盘上
    for ((id, elements) <- this.partitionedIterator) { // 将各个元素写到磁盘
      if (elements.hasNext) {
        for (elem <- elements) {
          writer.write(elem._1, elem._2)
        }
        val segment = writer.commitAndGet()
        lengths(id) = segment.length // 将各个分区的数据长度更新到lengths数组中
      }
    }
  }

  writer.close() // 关闭DiskBlockObjectWriter
```

```
    context.taskMetrics().incMemoryBytesSpilled(memoryBytesSpilled)
    context.taskMetrics().incDiskBytesSpilled(diskBytesSpilled)
    context.taskMetrics().incPeakExecutionMemory(peakMemoryUsedBytes)

    lengths // 返回lengths数组
  }
```

根据代码清单 8-68，writePartitionedFile 方法的执行步骤如下。

1）创建对每个分区的长度进行跟踪的数组 lengths。

2）调用 BlockManager 的 getDiskWriter 方法（见代码清单 6-83）获取 DiskBlockObject Writer。

3）如果 spills 中没有缓存溢出到磁盘的文件，即所有的数据依然都在内存中，那么执行以下操作。

① 获取当前使用的数据结构。根据对 ExternalSorter 的 insertAll 方法的分析，如果创建 ExternalSorter 时定义了聚合器，那么当前使用的数据结构是 PartitionedAppendOnlyMap，否则为 PartitionedPairBuffer。

② 调用 PartitionedAppendOnlyMap 或 PartitionedPairBuffer 的父类 WritablePartitioned PairCollection 的 destructiveSortedWritablePartitionedIterator 方法，得到对 PartitionedAppend OnlyMap 或 PartitionedPairBuffer 底层 data 数组中数据进行迭代的迭代器 WritablePartitioned Iterator。

③ 通过对 WritablePartitionedIterator 的迭代，将 PartitionedAppendOnlyMap 或 Partitioned PairBuffer 底层 data 数组中的数据按照分区 ID 分别写入到磁盘中，并且将各个分区的数据长度更新到 lengths 数组中。


注意 可以看到当没有溢出文件时，将 data 数组中的数据写入磁盘的方式与缓存溢出非常相似。

4）如果 spills 中缓存了溢出到磁盘的文件，即有些数据在内存中，有些数据已经溢出到了磁盘上，那么执行以下操作。

① 调用 partitionedIterator 方法（见代码清单 8-69），返回按照分区 ID 分组的迭代器。

② 对每个分区 ID 对应的迭代器进行迭代，将各个元素写到磁盘。

③ 将各个分区的数据长度更新到 lengths 数组中。

5）关闭 DiskBlockObjectWriter，更新任务度量信息，最后返回 lengths 数组。

ExternalSorter 的 partitionedIterator 方法（见代码清单 8-69）通过对集合按照指定的比较器进行排序，并且按照分区 ID 分组，生成迭代器。

代码清单8-69 partitionedIterator的实现

```
def partitionedIterator: Iterator[(Int, Iterator[Product2[K, C]])] = {
  val usingMap = aggregator.isDefined
  val collection: WritablePartitionedPairCollection[K, C] = if (usingMap) map else
```

```
        buffer
    if (spills.isEmpty) { // 如果spills中没有缓存溢出到磁盘的文件，即所有的数据依然都在内存中
      if (!ordering.isDefined) { // 对底层data数组中的数据只按照分区ID排序
        groupByPartition(destructiveIterator(collection.partitionedDestructiveSortedIterator
            (None)))
      } else { // 对底层data数组中的数据按照分区ID和key排序
        groupByPartition(destructiveIterator(
          collection.partitionedDestructiveSortedIterator(Some(keyComparator))))
      }
    } else { // 如果spills中缓存了溢出到磁盘的文件，即有些数据在内存中，有些数据已经溢出到了磁盘上
      merge(spills, destructiveIterator( // 将溢出的磁盘文件和data数组中的数据合并
        collection.partitionedDestructiveSortedIterator(comparator)))
    }
  }
```

根据代码清单 8-69，partitionedIterator 方法的执行步骤如下。

① 获取当前使用的数据结构。

② 如果 spills 中没有缓存溢出到磁盘的文件，即所有的数据依然都在内存中。如果定义了排序的 key，那么调用 PartitionedAppendOnlyMap 或 PartitionedPairBuffer 的 partitionedDestructiveSortedIterator 方法时，将对 PartitionedAppendOnlyMap 或 PartitionedPairBuffer 底层 data 数组中数据只按照分区 ID 排序，否则按照分区 ID 和 key 排序，然后调用 destructiveIterator 方法（见代码清单 8-70）返回迭代器，最后调用 groupByPartition 方法（见代码清单 8-71）对分区进行分组。

注意 由于 writePartitionedFile 方法在判断有溢出的磁盘文件时才调用 partitionedIterator 方法，因此条件 spills 不为空，所以不会执行这里介绍的程序逻辑。

③ 如果 spills 中缓存了溢出到磁盘的文件，即有些数据在内存中，有些数据已经溢出到了磁盘上，那么调用 PartitionedAppendOnlyMap 或 PartitionedPairBuffer 的 partitionedDestructiveSortedIterator 方法时，将对 PartitionedAppendOnlyMap 或 PartitionedPairBuffer 底层 data 数组中的数据按照分区 ID 和 key 排序，然后调用 destructiveIterator 方法（见代码清单 8-70）返回迭代器，最后调用 merge 方法（见代码清单 8-73）将溢出的磁盘文件和 data 数组中的数据合并。

代码清单8-70 destructiveIterator的实现

```
def destructiveIterator(memoryIterator: Iterator[((Int, K), C)]): Iterator[((Int,
    K), C)] = {
  if (isShuffleSort) {
    memoryIterator
  } else {
    readingIterator = new SpillableIterator(memoryIterator)
    readingIterator
  }
}
```

ExternalSorter 的 groupByPartition 方法（见代码清单 8-71）用于对 destructiveIterator

方法返回的迭代器按照分区 ID 分组。

代码清单8-71 groupByPartition的实现

```
private def groupByPartition(data: Iterator[((Int, K), C)])
    : Iterator[(Int, Iterator[Product2[K, C]])] =
{
  val buffered = data.buffered
  (0 until numPartitions).iterator.map(p => (p, new IteratorForPartition(p, buffered)))
}
```

根据代码清单 8-71，groupByPartition 方法实际给每个分区生成了一个 IteratorForPartition。但是每个分区对应的 IteratorForPartition 实例都使用了相同的数据源，即由 destructiveIterator 方法返回的迭代器，那么各个分区的数据该如何区分呢？让我们来看看 IteratorForPartition 的实现，如代码清单 8-72 所示。

代码清单8-72 IteratorForPartition的实现

```
private[this] class IteratorForPartition(partitionId: Int, data: BufferedIterator
    [((Int, K), C)])
  extends Iterator[Product2[K, C]]
{
  override def hasNext: Boolean = data.hasNext && data.head._1._1 == partitionId
  override def next(): Product2[K, C] = {
    if (!hasNext) {
      throw new NoSuchElementException
    }
    val elem = data.next()
    (elem._1._2, elem._2)
  }
}
```

根据代码清单 8-72，IteratorForPartition 的 hasNext 用于判断，对于指定的分区 ID 是否有下一个元素的条件如下。

- data 本身需要有下一个元素。
- data 的下一个元素对应的分区 ID 要与指定的分区 ID 一样。

有些读者可能会问，下一个元素对应的分区 ID 如果与指定的分区 ID 不一样，但是 data 中还有其他此分区的元素，怎么办？根据我们对 WritablePartitionedPairCollection 的 partitionedDestructiveSortedIterator 方法的实现分析，我们知道此时 data 中的元素已经按照分区 ID 排好序了，所以不会出现这种让人担忧的情况。

ExternalSorter 的 merge 方法（见代码清单 8-73）用于将 destructiveIterator 方法返回的可迭代访问内存中数据的迭代器与已经溢出到磁盘的文件进行合并。

代码清单8-73 数据合并

```
private def merge(spills: Seq[SpilledFile], inMemory: Iterator[((Int, K), C)])
    : Iterator[(Int, Iterator[Product2[K, C]])] = {
  val readers = spills.map(new SpillReader(_))
```

```
      val inMemBuffered = inMemory.buffered
      (0 until numPartitions).iterator.map { p =>
        val inMemIterator = new IteratorForPartition(p, inMemBuffered)
        val iterators = readers.map(_.readNextPartition()) ++ Seq(inMemIterator)
        if (aggregator.isDefined) {
          (p, mergeWithAggregation(
            iterators, aggregator.get.mergeCombiners, keyComparator, ordering.
                isDefined))
        } else if (ordering.isDefined) {
          (p, mergeSort(iterators, ordering.get))
        } else {
          (p, iterators.iterator.flatten)
        }
      }
    }
```

经过对 ExternalSorter 实现的 map 端输出持久化的分析，其处理步骤可以用图 8-15 表示。

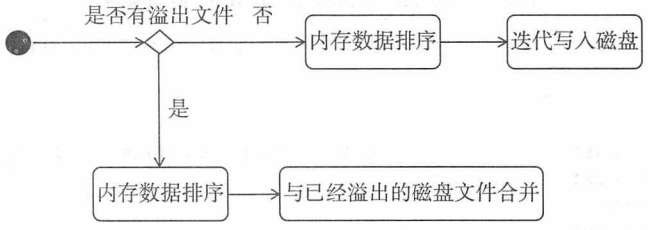


图 8-15　map 端输出的持久化

经过对 ExternalSorter 实现的 map 端输出的持久化过程的分析，我们知道持久化有以下两条路径。

- 有溢出文件时，将溢出的分区文件与内存中的数据进行合并后写入磁盘。
- 无溢出文件时，将内存中的数据进行整理、排序后写入磁盘。

无论哪种持久化执行过程，所有分区都会最终写入一个正式的 Block 文件，各个分区的数据按照分区 ID 和 key 进行排序输出到此文件中，所以每个 map 任务实际上最后只会生成一个磁盘文件，最终解决了 Spark 早期版本中一个 map 任务输出的 bucket 文件过多和磁盘 I/O 成为性能瓶颈的问题。此外，无论哪种排序方式，每输出完一个分区的中间结果时，都会使用 lengths 数组记录当前分区的长度，此长度将记录在分区索引文件中，以便下游 reduce 任务的读取。

经过对 SizeTracker、WritablePartitionedPairCollection、AppendOnlyMap、PartitionedPairBuffer、ExternalSorter 的深入理解，可以用图 8-16 来表示 ExternalSorter 实现的将 map 任务的中间结果输出到磁盘的整个过程。

图 8-16 展示了在指定了聚合函数后，map 任务中间结果输出的整体流程。map 任务按照先后顺序分别输出了（#,1）、（Apache,1）、（Spark,1）、（,1）、（Spark,1）、（is,1）等六条记录。在将数据更新到 AppendOnlyMap 前，计算每条记录的分区 ID。聚合的过程中可能不会产生

溢出文件，因此数据都将保留在 AppendOnlyMap 底层的 data 数组中，按照分区 ID 和 key 排序后持久化到文件中（图 8-16 中用红色箭头⊖表示不产生溢出文件的流程）。聚合的过程中可能会产生溢出文件，如果笔者将 spark.shuffle.spill.numElementsForceSpillThreshold 设置为 4，那么将产生一个溢出文件，再向 AppendOnlyMap 底层的 data 数组插入 ((1,is),1) 和 ((0,Spark),1) 两条记录后，由于未超过 4，因此不会溢出，最后将溢出文件与内存中的两条记录进行合并，生成数据文件（图 8-16 中用绿色箭头⊜表示产生溢出文件的流程）。

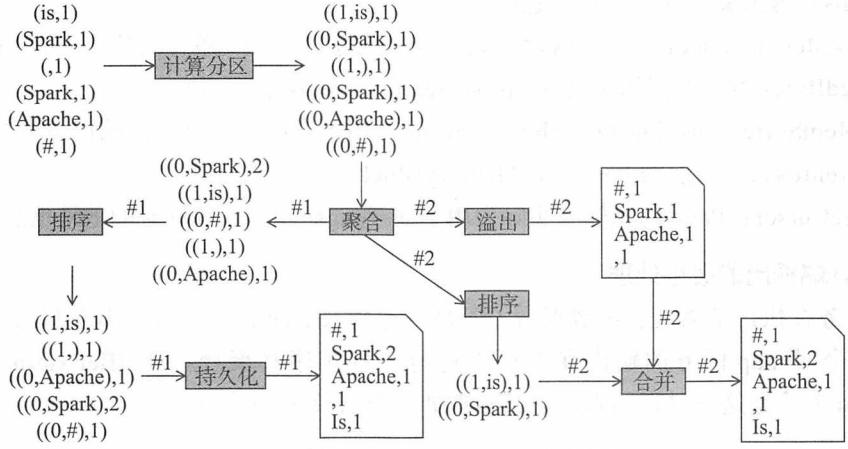


图 8-16 map 任务中间结果持久化的整体流程

图 8-16 采用了 AppendOnlyMap，对于 PartitionedPairBuffer 而言，只会更加简单，此处不再赘述。

8.11.2 ShuffleExternalSorter 详解

ShuffleExternalSorter 是专门用于对 Shuffle 数据进行排序的外部排序器，用于将 map 任务的输出存储到 Tungsten 中；在记录超过限制时，将数据溢出到磁盘。与 ExternalSorter 不同，ShuffleExternalSorter 本身并没有实现数据的持久化功能，具体的持久化将由 ShuffleExternalSorter 的调用者 UnsafeShuffleWriter 来实现。

ShuffleExternalSorter 包含的属性如下。

- numPartitions：分区数量。
- taskMemoryManager：即 TaskMemoryManager。
- blockManager：即 BlockManager。
- taskContext：即 TaskContext。
- writeMetrics：对 Shuffle 写入（也就是 map 任务输出到磁盘）的度量，即 Shuffle

⊖ 参见图 8-16 中标“#1”的箭头。
⊜ 参见图 8-16 中标“#2”的箭头。

WriteMetrics。

- numElementsForSpillThreshold：磁盘溢出的元素数量。可通过 spark.shuffle.spill.numElementsForceSpillThreshold 属性进行配置，默认为 1MB。
- fileBufferSizeBytes：创建的 DiskBlockObjectWriter 内部的文件缓冲大小。可通过 spark.shuffle.file.buffer 属性进行配置，默认是 32KB。
- allocatedPages：已经分配的 Page（即 MemoryBlock）列表。
- spills：溢出文件的元数据信息的列表。
- peakMemoryUsedBytes：内存中数据结构大小的峰值（单位为字节）。peakMemoryUsedBytes 方法专门用于返回 _peakMemoryUsedBytes 的值。
- inMemSorter：即 ShuffleInMemorySorter，用于在内存中对插入的记录进行排序。
- currentPage：当前的 Page（即 MemoryBlock）。
- pageCursor：Page 的光标。实际为用于向 Tungsten 写入数据时的地址信息。

1. map 端输出的缓存处理

map 任务在执行结束后会将数据写入磁盘，等待 reduce 任务获取。但在写入磁盘之前，Spark 可能会对 map 任务的输出在内存中进行一些排序和聚合。ShuffleExternalSorter 的 insertRecord 方法是这一过程的入口，其实现如代码清单 8-74 所示。

代码清单8-74 map端输出的缓存处理

```
public void insertRecord(Object recordBase, long recordOffset, int length, int
    partitionId)
  throws IOException {
  assert(inMemSorter != null);
  if (inMemSorter.numRecords() >= numElementsForSpillThreshold) {
    logger.info("Spilling data because number of spilledRecords crossed the
        threshold " +
      numElementsForSpillThreshold);
    spill(); // 将数据溢出到磁盘
  }
  // 检查是否有足够的空间将额外的记录插入到排序指针数组中，如果需要额外的空间，则增加数组的容量
  growPointerArrayIfNecessary();
  final int required = length + 4;
  // 检查是否有足够的空间，如果需要额外的空间，则申请分配新的Page
  acquireNewPageIfNecessary(required);
  assert(currentPage != null);
  final Object base = currentPage.getBaseObject();
  final long recordAddress = taskMemoryManager.encodePageNumberAndOffset(current
      Page, pageCursor);
  // 向Page所代表的内存块的起始地址写入数据的长度
  Platform.putInt(base, pageCursor, length);
  pageCursor += 4;
  // 将数据拷贝到Page中
  Platform.copyMemory(recordBase, recordOffset, base, pageCursor, length);
  pageCursor += length;
  // 将记录的元数据信息存储到内部用的长整型数组中
  inMemSorter.insertRecord(recordAddress, partitionId);
}
```

根据代码清单 8-74，insertRecord 方法的执行步骤如下。

① 如果 ShuffleInMemorySorter 中的记录数大于等于 numElementsForSpillThreshold，则调用 spill 方法（见代码清单 8-75）将数据溢出到磁盘。

② 调用 growPointerArrayIfNecessary 方法检查是否有足够的空间将额外的记录插入到排序指针数组中，如果需要额外的空间，则增加数组的容量。如果无法获取所需的空间，则内存中的数据将被溢出到磁盘。

③ 调用 acquireNewPageIfNecessary 方法检查是否有足够的空间（数据长度和 4 之和，这 4 字节用于在内存中存储数据的长度），如果需要额外的空间，则申请分配新的 Page。

④ 调用 TaskMemoryManager 的 encodePageNumberAndOffset 方法（见代码清单 8-22）返回页号和相对于内存块起始地址的偏移量（64 位长整型）。

⑤ 向 Page 所代表的内存块的起始地址写入数据的长度，长度将占用 4 个字节。这里表面上调用了 Platform 的 putInt 方法，实际调用了 sun.misc.UnSafe 的 putInt 方法。

⑥ 将 pageCursor 加 4。

⑦ 将数据拷贝到 Page 所代表的内存块中。这里表面上调用了 Platform 的 copyMemory 方法，实际调用了 sun.misc.UnSafe 的 copyMemory 方法。

⑧ 将数据的长度累加到 pageCursor。

⑨ 调用 ShuffleInMemorySorter 的 insertRecord 方法将记录的元数据信息存储到内部用的长整型数组中，以便于排序。其中高 24 位存储分区 ID，中间 13 位存储页号，低 27 位存储偏移量。

可以看到，ShuffleExternalSorter 对 map 端输出的缓存处理的实现与 ExternalSorter 非常相似，它们都将记录插入到内存。不同之处在于，ExternalSorter 除了简单的插入外，还有聚合的实现，而 ShuffleExternalSorter 没有；ExternalSorter 使用的是 JVM 的堆内存，而 ShuffleExternalSorter 使用的是 Tungsten 的内存（即有可能使用 JVM 的堆内存，也有可能使用操作系统的内存）。限于篇幅，ShuffleExternalSorter 的 growPointerArrayIfNecessary、acquireNewPageIfNecessary 及 ShuffleInMemorySorter 的 insertRecord 方法的实现留给感兴趣的读者自行了解。

2. 缓存溢出

根据之前的分析，当 ShuffleInMemorySorter 中的记录数大于等于 numElementsForSpillThreshold 时，则调用 MemoryConsumer 的模板方法 spill（见代码清单 8-27）将数据溢出到磁盘。模板方法 spill 实际调用了 ShuffleExternalSorter 实现的 spill 方法（见代码清单 8-75）。

代码清单8-75 缓存溢出

```
@Override
public long spill(long size, MemoryConsumer trigger) throws IOException {
  if (trigger != this || inMemSorter == null || inMemSorter.numRecords() == 0) {
    return 0L;
  }
```

```
    logger.info("Thread {} spilling sort data of {} to disk ({} {} so far)",
      Thread.currentThread().getId(),
      Utils.bytesToString(getMemoryUsage()),
      spills.size(),
      spills.size() > 1 ? " times" : " time");

    writeSortedFile(false); // 将内存中的记录进行排序后输出到磁盘
    final long spillSize = freeMemory(); // 将所使用的Page（即MemoryBlock）全部释放
    inMemSorter.reset(); // 重置ShuffleInMemorySorter底层的长整型数组，以便于下次排序
    taskContext.taskMetrics().incMemoryBytesSpilled(spillSize);
    return spillSize; // 返回溢出的数据大小
  }
```

根据代码清单 8-75，spill 方法的执行步骤如下。

1）调用 writeSortedFile 方法将内存中的记录进行排序后输出到磁盘。这里用到的排序方式有两种：一种是对分区 ID 进行比较的排序，一种采用了基数排序（Radix Sort）。

2）由于内存中的记录已经溢出到磁盘，因此调用 freeMemory 方法将所使用的 Page（即 MemoryBlock）全部释放。

3）重置 ShuffleInMemorySorter 底层的长整型数组，以便于下次排序。

4）更新任务度量信息。

5）返回溢出的数据大小。

可以看到，ShuffleExternalSorter 对溢出的实现与 ExternalSorter 非常相似，都在溢出前进行了排序，也都有按照分区 ID 进行排序的实现。不同之处在于，ExternalSorter 有按照分区 ID 和 key 进行排序，而 ShuffleExternalSorter 有基于基数排序（Radix Sort）的实现。限于篇幅，writeSortedFile 方法的实现留给感兴趣的读者自行了解，笔者相信有了对 ExternalSorter 的缓存溢出的了解，读者理解 writeSortedFile 方法的实现也没有问题。

8.12 Shuffle 管理器

ShuffleManager 本身依赖于存储体系，但由于其功能与计算更为紧密，所以将它视为计算引擎的一部分。根据 ShuffleManager 的名字，就知道它的主要功能是对 Shuffle 进行管理。早在 5.7 节介绍存储体系的构建时，就对 ShuffleManager 的实例化进行了一些介绍，因此我们发现，无论 spark.shuffle.manager 属性是 sort 还是 tungsten-sort，都是对 ShuffleManager 的实现类 SortShuffleManager 的实例化。特质 ShuffleManager 目前只有 SortShuffleManager 这一个实现类[⊖]，因此笔者将直接对 SortShuffleManager 进行分析。

⊖ 在 Spark 2.0.0 版本以前，ShuffleManager 还有另一个实现类 HashShuffleManager。由于 HashShuffleManager 在 Shuffle 过程中随着 map 任务数量或者 reduce 任务数量的增加，基于 Hash 的 Shuffle 在性能上的表现相比基于 Sort 的 Shuffle 越来越差，因此 Spark 2.0.0 版本移除了 HashShuffleManager。笔者在《深入理解 Spark》一书中对 HashShuffleManager 有详细的介绍，感兴趣的读者可以阅读此书。

8.12.1 ShuffleWriter 详解

SortShuffleManager 依赖于 ShuffleWriter 提供的服务，抽象类 ShuffleWriter 定义了将 map 任务的中间结果输出到磁盘上的功能规范，包括将数据写入磁盘和关闭 ShuffleWriter。ShuffleWriter 的定义如下。

```
private[spark] abstract class ShuffleWriter[K, V] {
  @throws[IOException]
  def write(records: Iterator[Product2[K, V]]): Unit
  def stop(success: Boolean): Option[MapStatus]
}
```

ShuffleWriter 定义的 write 方法用于将 map 任务的结果写到磁盘，而 stop 方法可以关闭 ShuffleWriter。

ShuffleWriter 一共有三个子类，分别为 SortShuffleWriter、UnsafeShuffleWriter 及 BypassMergeSortShuffleWriter，本节将逐一介绍。

1. ShuffleHandle 详解

ShuffleHandle 是不透明的 Shuffle 句柄，ShuffleManager 使用它向 Task 传递 Shuffle 信息。由于 SortShuffleWriter 依赖于 ShuffleHandle 的实现，因此我们需要先分析 ShuffleHandle。ShuffleHandle 的定义如下。

```
@DeveloperApi
abstract class ShuffleHandle(val shuffleId: Int) extends Serializable {}
```

BaseShuffleHandle 是 ShuffleHandle 的直接子类，仅用于 ShuffleManager 的 registerShuffle 方法的参数。BaseShuffleHandle 的定义如下。

```
private[spark] class BaseShuffleHandle[K, V, C](
    shuffleId: Int,
    val numMaps: Int,
    val dependency: ShuffleDependency[K, V, C])
  extends ShuffleHandle(shuffleId)
```

BaseShuffleHandle 有 BypassMergeSortShuffleHandle 和 SerializedShuffleHandle 两个子类。

```
private[spark] class SerializedShuffleHandle[K, V](
  shuffleId: Int,
  numMaps: Int,
  dependency: ShuffleDependency[K, V, V])
  extends BaseShuffleHandle(shuffleId, numMaps, dependency) {
}
private[spark] class BypassMergeSortShuffleHandle[K, V](
  shuffleId: Int,
  numMaps: Int,
  dependency: ShuffleDependency[K, V, V])
  extends BaseShuffleHandle(shuffleId, numMaps, dependency) {
}
```

SerializedShuffleHandle 用于确定何时选择使用序列化的 Shuffle，BypassMergeSort ShuffleHandle 则用于确定何时选择绕开合并和排序的 Shuffle 路径。

2. MapStatus 详解

MapStatus 用于表示 ShuffleMapTask 返回给 TaskScheduler 的执行结果。MapStatus 的定义如下。

```
private[spark] sealed trait MapStatus {
  def location: BlockManagerId
  def getSizeForBlock(reduceId: Int): Long
}
```

根据上述代码，location 方法用于返回 ShuffleMapTask 运行的位置，即所在节点的 BlockManager 的身份标识 BlockManagerId。getSizeForBlock 用于返回 reduce 任务需要拉取的 Block 的大小（单位为字节）。

MapStatus 的伴生对象中定义了 apply 函数（见代码清单 8-76），按照 Scala 的语言特性，我们可以直接使用 MapStatus(BlockManagerId, partitionLengths) 这样的形式创建 MapStatus 实例。

代码清单8-76 MapStatus伴生对象的apply函数

```
def apply(loc: BlockManagerId, uncompressedSizes: Array[Long]): MapStatus = {
  if (uncompressedSizes.length > 2000) {
    HighlyCompressedMapStatus(loc, uncompressedSizes)
  } else {
    new CompressedMapStatus(loc, uncompressedSizes)
  }
}
```

根据代码清单 8-76，apply 函数根据 uncompressedSizes 的长度是否大于 2000，分别创建 HighlyCompressedMapStatus 和 CompressedMapStatus，这说明对于较大的数据量使用高度压缩的 HighlyCompressedMapStatus，一般的数据量则使用 CompressedMapStatus。

3. SortShuffleWriter 的实现

SortShuffleWriter 是 ShuffleWriter 的实现类之一，提供了对 Shuffle 数据的排序功能。SortShuffleWriter 使用 ExternalSorter 作为排序器，由于 ExternalSorter 底层使用了 PartitionedAppendOnlyMap 和 PartitionedPairBuffer 两种缓存，因此 SortShuffleWriter 还支持对 Shuffle 数据的聚合功能。

在 SortShuffleWriter 中，包含以下属性。

- shuffleBlockResolver：即 IndexShuffleBlockResolver。
- handle：即 BaseShuffleHandle。
- mapId：map 任务的身份标识。
- context：即任务上下文（TaskContext），由于 TaskContext 只有 TaskContextImpl 这一个实现类，因此我们也可以认为是 TaskContextImpl。

- dep：handle（即 BaseShuffleHandle）的 dependency 属性（类型为 ShuffleDependency）。
- blockManager：即 SparkEnv 的子组件 BlockManager。
- sorter：即 ExternalSorter。
- stopping：是否正在停止。
- mapStatus：map 任务的状态，即 MapStatus。
- writeMetrics：对 Shuffle 写入（也就是 map 任务输出到磁盘）的度量，即 ShuffleWriteMetrics。

SortShuffleWriter 的核心实现在于将 map 任务的输出结果写到磁盘的 write 方法，其实现如代码清单 8-77 所示。

代码清单8-77 SortShuffleWriter的write方法

```
override def write(records: Iterator[Product2[K, V]]): Unit = {
  sorter = if (dep.mapSideCombine) { // 创建ExternalSorter
    require(dep.aggregator.isDefined, "Map-side combine without Aggregator
        specified!")
    new ExternalSorter[K, V, C](
      context, dep.aggregator, Some(dep.partitioner), dep.keyOrdering, dep.
          serializer)
  } else {
    new ExternalSorter[K, V, V](
      context, aggregator = None, Some(dep.partitioner), ordering = None, dep.
          serializer)
  }
  sorter.insertAll(records) // 将map任务的输出记录插入到缓存中
  // 获取Shuffle数据文件
  val output = shuffleBlockResolver.getDataFile(dep.shuffleId, mapId)
  val tmp = Utils.tempFileWith(output)
  try { // 将map端缓存的数据写入到磁盘中，并生成Block文件对应的索引文件
    val blockId = ShuffleBlockId(dep.shuffleId, mapId, IndexShuffleBlockResolver.
        NOOP_REDUCE_ID)
    val partitionLengths = sorter.writePartitionedFile(blockId, tmp)
    shuffleBlockResolver.writeIndexFileAndCommit(dep.shuffleId, mapId, partition
        Lengths, tmp)
    // 构造并返回MapStatus
    mapStatus = MapStatus(blockManager.shuffleServerId, partitionLengths)
  } finally {
    if (tmp.exists() && !tmp.delete()) {
      logError(s"Error while deleting temp file ${tmp.getAbsolutePath}")
    }
  }
}
```

根据代码清单 8-77，write 方法的执行步骤如下。

1）如果 ShuffleDependency 的 mapSideCombine 属性为 true（即允许在 map 端进行合并），那么创建 ExternalSorter 时，将 ShuffleDependency 的 aggregator 和 keyOrdering 传递给 ExternalSorter 的 aggregator 和 ordering 属性，否则不进行传递。这也间接决定了 External Sorter 选择 PartitionedAppendOnlyMap 还是 PartitionedPairBuffer。

2）调用 ExternalSorter 的 insertAll 方法（已在 8.11.1 节介绍）将 map 任务的输出记录插入到缓存中。

3）调用 IndexShuffleBlockResolver 的 getDataFile 方法（见代码清单 8-38）获取 Shuffle 数据文件。

4）构造 ShuffleBlockId。

5）调用 ExternalSorter 的 writePartitionedFile 方法（已在 8.11.1 节介绍）将 map 端缓存的数据写入到磁盘中。partitionLengths 即为 writePartitionedFile 方法返回的 lengths 数组，其中记录了各个分区的长度。

6）调用 IndexShuffleBlockResolver 的 writeIndexFileAndCommit 方法（见代码清单 8-41）生成 Block 文件对应的索引文件。此索引文件用于记录各个分区在 Block 文件中对应的偏移量，以便于 reduce 任务拉取时使用。

7）构造 MapStatus。

笔者虽然对 SortShuffleWriter 进行了详细的源码分析，但为了带给读者更加直观的感受，我们指定 ShuffleDependency 的 mapSideCombine 属性为 true，并且设置了聚合函数，那么 SortShuffleWriter 的 write 方法的执行流程就可以用图 8-17 来表示。

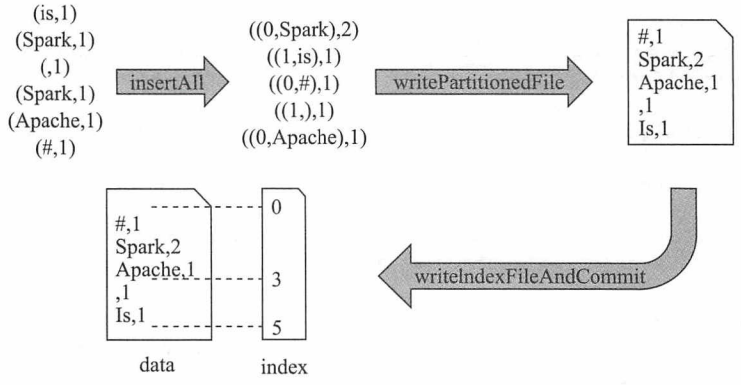


图 8-17　SortShuffleWriter 在 map 端聚合的执行流程

图 8-17 展示了 map 任务结果更新到内存（即调用 ExternalSorter 的 insertAll 方法时将选择 PartitionedAppendOnlyMap）、将内存中的数据持久化到分区数据文件（即调用 ExternalSorter 的 writePartitionedFile）、给分区数据文件生成分区索引文件（即调用 IndexShuffleBlockResolver 的 writeIndexFileAndCommit）的一系列动作。

当指定 ShuffleDependency 的 mapSideCombine 属性为 false 时，SortShuffleWriter 的 write 方法的执行流程如图 8-18 所示。

图 8-18 展示了将 map 任务结果保存到内存（即调用 ExternalSorter 的 insertAll 方法时将选择 PartitionedPairBuffer）、将内存中的数据持久化到分区数据文件（即调用 ExternalSorter 的 writePartitionedFile）、给分区数据文件生成分区索引文件（即调用 IndexShuffleBlock

Resolver 的 writeIndexFileAndCommit）等动作。

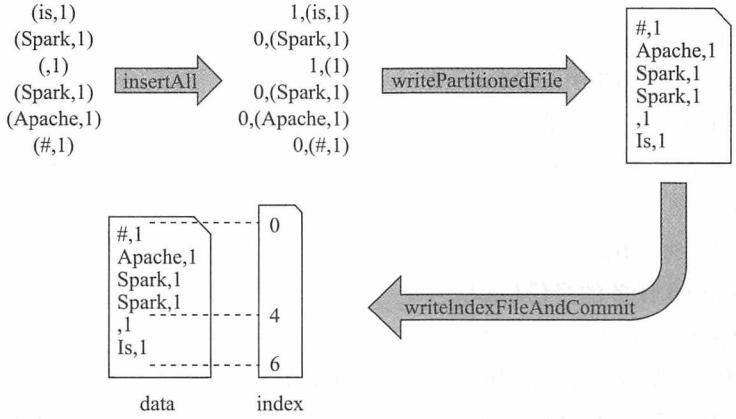


图 8-18 SortShuffleWriter 不在 map 端聚合的执行流程

SortShuffleWriter 的伴生对象中提供了用于判断是否绕开合并和排序的 shouldBypass MergeSort 方法，其实现如代码清单 8-78 所示。

代码清单8-78 ShouldByPassMergeSort方法

```
def shouldBypassMergeSort(conf: SparkConf, dep: ShuffleDependency[_, _, _]): Boolean
    = {
  if (dep.mapSideCombine) {
    require(dep.aggregator.isDefined, "Map-side combine without Aggregator
        specified!")
    false
  } else {
    val bypassMergeThreshold: Int = conf.getInt("spark.shuffle.sort.bypassMerge
        Threshold", 200)
    dep.partitioner.numPartitions <= bypassMergeThreshold
  }
}
```

根据代码清单 8-78，如果 ShuffleDependency 的 mapSideCombine 属性为 false，且 Shuffle Dependency 的分区计算器中的分区数量小于等于 bypassMergeThreshold，返回 true，否则返回 false。bypassMergeThreshold 可通过 spark.shuffle.sort.bypassMergeThreshold 属性配置，默认为 200。

4. BypassMergeSortShuffleWriter 的实现

有时候，map 端不需要在持久化数据之前进行聚合、排序等操作，那么 ShuffleWriter 的实现类之一 BypassMergeSortShuffleWriter（从命名可以看出这是绕开合并、排序的 Shuffle Writer）就可以派上用场。

BypassMergeSortShuffleWriter 包含以下属性。

- fileBufferSize：文件缓冲大小。可通过 spark.shuffle.file.buffer 属性配置，默认为 32KB。

- transferToEnabled：是否采用 NIO 的从文件流到文件流的复制方式。可通过 spark.file.transferTo 属性配置，默认为 true。
- numPartitions：分区数。
- blockManager：即 BlockManager。
- partitioner：分区计算器（Partitioner）。
- writeMetrics：对 Shuffle 写入（也就是 map 任务输出到磁盘）的度量，即 ShuffleWriteMetrics。
- shuffleId：Shuffle 的唯一标识。
- mapId：map 任务的身份标识。
- serializer：序列化器（Serializer）。
- shuffleBlockResolver：即 IndexShuffleBlockResolver。
- partitionWriters：DiskBlockObjectWriter 类型的数组，每一个 DiskBlockObjectWriter 处理一个分区的数据。
- partitionWriterSegments：FileSegment 的数组，每一个 FileSegment 对应一个 DiskBlockObjectWriter 处理的文件片。
- mapStatus：map 任务的状态，即 MapStatus。
- partitionLengths：长整型数组，每个元素记录一个分区的数据长度。
- stopping：标记 BypassMergeSortShuffleWriter 是否正在停止中。

BypassMergeSortShuffleWriter 的核心实现在于将 map 任务的输出结果写到磁盘的 write 方法，其实现如代码清单 8-79 所示。

代码清单8-79　BypassMergeSortShuffleWriter的write方法

```
@Override
public void write(Iterator<Product2<K, V>> records) throws IOException {
  assert (partitionWriters == null);
  if (!records.hasNext()) { // 如果没有输出的记录，则只生成索引文件
    partitionLengths = new long[numPartitions];
    shuffleBlockResolver.writeIndexFileAndCommit(shuffleId, mapId, partitionLengths,
        null);
    mapStatus = MapStatus$.MODULE$.apply(blockManager.shuffleServerId(), partition
        Lengths);
    return;
  }
  final SerializerInstance serInstance = serializer.newInstance();
  final long openStartTime = System.nanoTime();
  partitionWriters = new DiskBlockObjectWriter[numPartitions];
  partitionWriterSegments = new FileSegment[numPartitions];
  for (int i = 0; i < numPartitions; i++) {
    final Tuple2<TempShuffleBlockId, File> tempShuffleBlockIdPlusFile =
      // 给每个分区创建分区数据待写入的文件
      blockManager.diskBlockManager().createTempShuffleBlock();
    final File file = tempShuffleBlockIdPlusFile._2();
    final BlockId blockId = tempShuffleBlockIdPlusFile._1();
```

```
    partitionWriters[i] = // 创建向此文件写入的DiskBlockObjectWriter
      blockManager.getDiskWriter(blockId, file, serInstance, fileBufferSize,
          writeMetrics);
  }
  writeMetrics.incWriteTime(System.nanoTime() - openStartTime);

  while (records.hasNext()) { // 向临时Shuffle文件的输出流中写入键值对
    final Product2<K, V> record = records.next();
    final K key = record._1();
    partitionWriters[partitioner.getPartition(key)].write(key, record._2());
  }

  for (int i = 0; i < numPartitions; i++) {
    final DiskBlockObjectWriter writer = partitionWriters[i];
    // 将临时Shuffle文件的输出流中的数据写入到磁盘
    partitionWriterSegments[i] = writer.commitAndGet();
    writer.close();
  }

  // 获取Shuffle数据文件
  File output = shuffleBlockResolver.getDataFile(shuffleId, mapId);
  File tmp = Utils.tempFileWith(output);
  try {
    // 将每个分区的文件合并到Shuffle数据文件中
    partitionLengths = writePartitionedFile(tmp);
    // 生成Block文件对应的索引文件
    shuffleBlockResolver.writeIndexFileAndCommit(shuffleId, mapId, partitionLengths,
        tmp);
  } finally {
    if (tmp.exists() && !tmp.delete()) {
      logger.error("Error while deleting temp file {}", tmp.getAbsolutePath());
    }
  }
  mapStatus = MapStatus$.MODULE$.apply(blockManager.shuffleServerId(), partitionLengths);
}
```

根据代码清单 8-79，BypassMergeSortShuffleWriter 的 write 方法的执行步骤如下。

1）如果没有输出的记录，则首先创建 partitionLengths 数组（数组的长度为 numPartitions），然后调用 IndexShuffleBlockResolver 的 writeIndexFileAndCommit 方法生成索引文件（此时创建的索引文件中只有 0 这一个偏移量），最后创建 MapStatus。

2）创建 partitionWriters 和 partitionWriterSegments 数组。

3）给每个分区创建分区数据待写入的文件，并调用 BlockManager 的 getDiskWriter 方法（见代码清单 6-83）创建向此文件写入的 DiskBlockObjectWriter。

4）迭代待输出的记录，使用分区计算器并通过每条记录的 key，获取记录的分区 ID，调用此分区 ID 对应的 DiskBlockObjectWriter 的 write 方法（见代码清单 6-114），向临时 Shuffle 文件的输出流中写入键值对。

5）调用每个分区对应的 DiskBlockObjectWriter 的 commitAndGet 方法（见代码清单 6-115），将临时 Shuffle 文件的输出流中的数据写入到磁盘，并将返回的 FileSegment 放入

partitionWriterSegments 数组中，以此分区 ID 为索引的位置。

6）调用 IndexShuffleBlockResolver 的 getDataFile 方法（见代码清单 8-38），获取 Shuffle 数据文件。

7）调用 BypassMergeSortShuffleWriter 的 writePartitionedFile 方法（见代码清单 8-80），将每个分区的文件合并到 Shuffle 数据文件中。

8）调用 IndexShuffleBlockResolver 的 writeIndexFileAndCommit 方法（见代码清单 8-41），生成 Block 文件对应的索引文件。此索引文件用于记录各个分区在 Block 文件中对应的偏移量，以便于 reduce 任务拉取时使用。

9）构造并返回 MapStatus。

代码清单8-80　生成分区文件

```
private long[] writePartitionedFile(File outputFile) throws IOException {
  final long[] lengths = new long[numPartitions];
  if (partitionWriters == null) {
    return lengths;
  }
  // 打开正式输出的文件输出流
  final FileOutputStream out = new FileOutputStream(outputFile, true);
  final long writeStartTime = System.nanoTime();
  boolean threwException = true;
  try {
    for (int i = 0; i < numPartitions; i++) {
      final File file = partitionWriterSegments[i].file();
      if (file.exists()) {
        final FileInputStream in = new FileInputStream(file);
        boolean copyThrewException = true;
        try { // 将输入流中的字节拷贝到输出流中
          lengths[i] = Utils.copyStream(in, out, false, transferToEnabled);
          copyThrewException = false;
        } finally {
          Closeables.close(in, copyThrewException);
        }
        if (!file.delete()) {
          logger.error("Unable to delete file for partition {}", i);
        }
      }
    }
    threwException = false;
  } finally {
    Closeables.close(out, threwException);
    writeMetrics.incWriteTime(System.nanoTime() - writeStartTime);
  }
  partitionWriters = null;
  return lengths;
}
```

根据代码清单 8-80，BypassMergeSortShuffleWriter 的 writePartitionedFile 方法的执行步骤如下。

1）创建长整型数组 lengths，大小为分区数。

2）打开正式输出的文件输出流。

3）打开 partitionWriterSegments 中每个分区 ID 对应的文件的输入流，调用 Utils 工具类的 copyStream 方法（参阅附录 A）将输入流中的字节拷贝到输出流中。由于遍历分区 ID 是从 0 开始的，因此最后写入分区数据文件的数据也是按照分区 ID 排好序的。Utils 工具类的 copyStream 方法将返回拷贝的字节数，这个字节数保存在 lengths 数组的对应分区的位置。

4）返回 lengths。

经过对 BypassMergeSortShuffleWriter 的分析，可以用图 8-19 来表示它的执行流程。

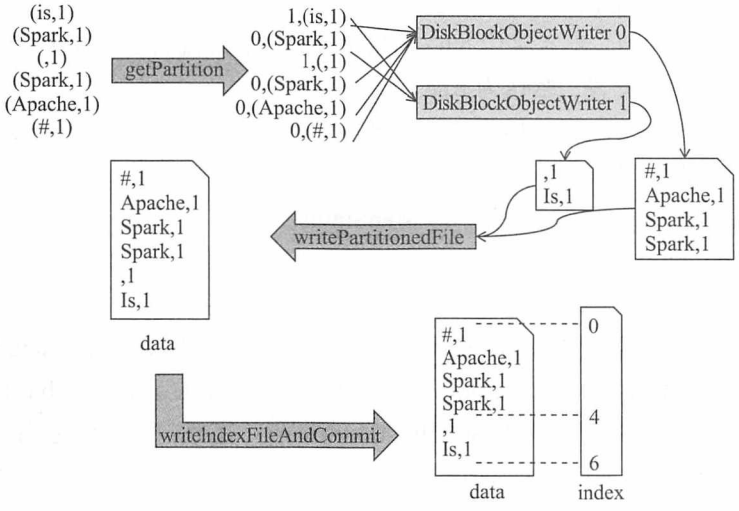


图 8-19 BypassMergeSortShuffleWriter 的 write 方法的执行流程

图 8-19 展示了给输入记录的 key 计算分区 ID，并给每个分区 ID 指定一个 DiskBlock ObjectWriter，将此分区的记录写入到临时 Shuffle 文件，然后调用 BypassMergeSortShuffle Writer 的 writePartitionedFile 方法，将所有的临时 Shuffle 文件按照分区 ID 升序写入正式的 Shuffle 数据文件，最后调用 IndexShuffleBlockResolver 的 writeIndexFile AndCommit 方法创建 Shuffle 索引文件。

5. UnsafeShuffleWriter 的实现

UnsafeShuffleWriter 是 ShuffleWriter 的实现类之一，底层使用 ShuffleExternalSorter 作为外部排序器，所以 UnsafeShuffleWriter 不具备 SortShuffleWriter 的聚合功能。UnsafeShuffle Writer 将使用 Tungsten 的内存作为缓存，以提高写入磁盘的性能。

UnsafeShuffleWriter 包含以下属性。

- blockManager：即 BlockManager。
- shuffleBlockResolver：即 IndexShuffleBlockResolver。
- memoryManager：即 TaskMemoryManager。

- serializer：序列化器的实例。
- partitioner：即 Partitioner。
- writeMetrics：对 Shuffle 写入（也就是 map 任务输出到磁盘）的度量，即 ShuffleWriteMetrics。
- shuffleId：Shuffle 的唯一标识。
- mapId：map 任务的身份标识。
- taskContext：即 TaskContext。
- sparkConf：即 SparkConf。
- transferToEnabled：是否采用 NIO 的从文件流到文件流的复制方式。可通过 spark.file.transferTo 属性配置，默认为 true。
- initialSortBufferSize：初始化的排序缓冲大小。可通过 spark.shuffle.sort.initialBufferSize 属性配置，默认为 4096。
- mapStatus：map 任务的状态，即 MapStatus。
- sorter：即 ShuffleExternalSorter。
- peakMemoryUsedBytes：使用内存的峰值（单位为字节）。
- serBuffer：类型为 MyByteArrayOutputStream，实际是 ByteArrayOutputStream 的子类，并提供了暴露 ByteArrayOutputStream 内部存储数据的字节数组 buf 的 getBuf 方法。
- serOutputStream：类型为 SerializationStream（是用于序列化对象的输出流），serOutputStream 实际是将 serBuffer 包装为 SerializationStream 后的对象。
- stopping：标记 UnsafeShuffleWriter 是否正在停止中。

UnsafeShuffleWriter 的核心实现在于将 map 任务的输出结果写到磁盘的 write 方法，其实现如代码清单 8-81 所示。

代码清单8-81 UnsafeShuffleWriter的write方法

```
@Override
public void write(scala.collection.Iterator<Product2<K, V>> records) throws
    IOException {
  boolean success = false;
  try {
    while (records.hasNext()) {
      insertRecordIntoSorter(records.next()); // 将记录插入排序器
    }
    closeAndWriteOutput(); // 将map任务输出的数据持久化到磁盘
    success = true;
  } finally {
    if (sorter != null) {
      try {
        sorter.cleanupResources();
      } catch (Exception e) {
        if (success) {
          throw e;
        } else {
```

```
                logger.error("In addition to a failure during writing, we failed during " +
                        "cleanup.", e);
            }
        }
    }
  }
}
```

根据代码清单 8-81，UnsafeShuffleWriter 的 write 方法的执行步骤如下。

1）迭代输入的每条记录，并调用 insertRecordIntoSorter 方法（见代码清单 8-82）将记录插入排序器。

2）调用 closeAndWriteOutput 方法（见代码清单 8-83）将 map 任务输出的数据持久化到磁盘。

代码清单8-82 将记录插入排序器

```
void insertRecordIntoSorter(Product2<K, V> record) throws IOException {
  assert(sorter != null);
  final K key = record._1();
  final int partitionId = partitioner.getPartition(key); // 计算记录的分区ID
  serBuffer.reset();
  serOutputStream.writeKey(key, OBJECT_CLASS_TAG);
  serOutputStream.writeValue(record._2(), OBJECT_CLASS_TAG);
  serOutputStream.flush();
  final int serializedRecordSize = serBuffer.size();
  assert (serializedRecordSize > 0);
  sorter.insertRecord( // 将serBuffer底层的字节数组插入到Tungsten的内存中
    serBuffer.getBuf(), Platform.BYTE_ARRAY_OFFSET, serializedRecordSize, partitionId);
}
```

根据代码清单 8-82，insertRecordIntoSorter 方法的执行步骤如下。

1）计算记录的分区 ID。

2）重置 serBuffer。

3）将记录写入到 serOutputStream 中。

4）调用 ShuffleExternalSorter 的 insertRecord 方法，将 serBuffer 底层的字节数组插入到 Tungsten 的内存中。

代码清单8-83 closeAndWriteOutput方法的实现

```
void closeAndWriteOutput() throws IOException {
  assert(sorter != null);
  updatePeakMemoryUsed(); // 更新使用内存的峰值
  serBuffer = null;
  serOutputStream = null;
  final SpillInfo[] spills = sorter.closeAndGetSpills(); // 获得溢出的文件信息数组
  sorter = null;
  final long[] partitionLengths;
  // 获取正式的输出数据文件
  final File output = shuffleBlockResolver.getDataFile(shuffleId, mapId);
  final File tmp = Utils.tempFileWith(output);
```

```
    try {
      try {
        partitionLengths = mergeSpills(spills, tmp); // 合并所有溢出文件到正式的输出数据文件
      } finally {
        for (SpillInfo spill : spills) {
          if (spill.file.exists() && ! spill.file.delete()) {
            logger.error("Error while deleting spill file {}", spill.file.getPath());
          }
        }
      } // 创建索引文件
      shuffleBlockResolver.writeIndexFileAndCommit(shuffleId, mapId, partitionLengths,
          tmp);
    } finally {
      if (tmp.exists() && !tmp.delete()) {
        logger.error("Error while deleting temp file {}", tmp.getAbsolutePath());
      }
    }
    mapStatus = MapStatus$.MODULE$.apply(blockManager.shuffleServerId(), partitionLengths);
  }
```

根据代码清单 8-83，closeAndWriteOutput 方法的执行步骤如下。

1）更新使用内存的峰值。

2）关闭 ShuffleExternalSorter，并获得溢出的文件信息数组。

3）调用 IndexShuffleBlockResolver 的 getDataFile 方法，获取正式的输出数据文件。

4）合并所有溢出文件到正式的输出数据文件。限于篇幅，mergeSpills 方法的实现这里就不多介绍了。

5）调用 IndexShuffleBlockResolver 的 writeIndexFileAndCommit 方法创建索引文件。

6）构造并返回 MapStatus。

根据对 UnsafeShuffleWriter 的分析，可以看到 UnsafeShuffleWriter 的执行过程与图 8-18 所表示的 SortShuffleWriter 在不允许 map 端合并（即 ShuffleDenpendency 的 mapSideCombine 属性为 false）情况下的执行过程非常类似，只不过 SortShuffleWriter 底层的 PartitionedPairBuffer 使用的是 JVM 的内存，而 UnsafeShuffleWriter 使用的则是 Tungsten（既有可能是 JVM 内存，也有可能是操作系统内存）。

8.12.2 ShuffleBlockFetcherIterator 详解

ShuffleBlockFetcherIterator 是用于获取多个 Block 的迭代器。如果 Block 在本地，那么从本地的 BlockManager 获取；如果 Block 在远端，那么通过 ShuffleClient 请求远端节点上的 BlockTransferService 获取。

ShuffleBlockFetcherIterator 中有很多属性，分别如下。

- context：当前 Task 的上下文，即 TaskContext。
- shuffleClient：即 ShuffleClient。此 ShuffleClient 将用于从远端节点下载 Block。
- blockManager：本地的 BlockManager。

- blocksByAddress：将要获取的 Block 与所在地址的关系，类型为 Seq[(BlockManagerId, Seq[(BlockId, Long)])]。从此属性可以看出，每个 BlockManager 中包含一到多个任务需要的 Block。
- maxBytesInFlight：单次航班（即一批请求）请求的最大字节数。这批请求的字节总数不能超过 maxBytesInFlight，而且每个请求的字节数不能超过 maxBytesInFlight 的 1/5。可以通过参数 spark.reducer.maxMbInFlight 来控制大小（默认为 48MB）。为什么每个请求的字节数不能超过 maxBytesInFlight 的 1/5？这样做是为了提高请求的并发度，保证至少向 5 个不同的节点发送请求获取数据，最大限度地利用各节点的资源。
- maxReqsInFlight：单次航班的最多请求数。此参数可以通过 spark.reducer.maxReqsInFlight 属性配置，默认为 Integer.MAX_VALUE。
- numBlocksToFetch：一共要获取的 Block 数量。
- numBlocksProcessed：已经处理的 Block 数量。
- startTime：ShuffleBlockFetcherIterator 的启动时间。
- localBlocks：缓存了本地 BlockManager 管理的 Block 的 BlockId。
- remoteBlocks：缓存了远端 BlockManager 管理的 Block 的 BlockId。
- results：用于保存获取 Block 的结果信息（FetchResult）。FetchResult 的实现如下。

```
private[storage] sealed trait FetchResult {
  val blockId: BlockId
  val address: BlockManagerId
}
```

- currentResult：当前正在处理的 FetchResult。
- fetchRequests：获取 Block 的请求信息（FetchRequest）的队列。FetchRequest 的实现如下。

```
case class FetchRequest(address: BlockManagerId, blocks: Seq[(BlockId, Long)]) {
  val size = blocks.map(_._2).sum
}
```

FetchRequest 的 size 属性返回 FetchRequest 要下载的所有 Block 的大小之和。

- bytesInFlight：当前航班（批次）的请求的字节数。
- reqsInFlight：当前航班（批次）的请求的数量。
- shuffleMetrics：Shuffle 的度量信息。
- isZombie：ShuffleBlockFetcherIterator 是否处于激活状态。如果 isZombie 为 true，则 ShuffleBlockFetcherIterator 处于非激活状态。

ShuffleBlockFetcherIterator 中提供了很多方法，本节将对最为重要的初始化、划分本地与远程 Block、获取远端 Block、获取本地 Block 等方法进行介绍。

1. 初始化

构造 ShuffleBlockFetcherIterator 的时候会调用 initialize 方法（见代码清单 8-84），对 Shuffle

BlockFetcherIterator 进行初始化。

代码清单8-84 ShuffleBlockFetcherIterator的初始化

```
private[this] def initialize(): Unit = {
  // 给TaskContextImpl添加任务完成的监听器
  context.addTaskCompletionListener(_ => cleanup())
  val remoteRequests = splitLocalRemoteBlocks() // 划分从本地读取和需要远程读取的Block的
      请求
  fetchRequests ++= Utils.randomize(remoteRequests) // 将FetchRequest随机排序后存入
      fetchRequests
  assert ((0 == reqsInFlight) == (0 == bytesInFlight),
    "expected reqsInFlight = 0 but found reqsInFlight = " + reqsInFlight +
    ", expected bytesInFlight = 0 but found bytesInFlight = " + bytesInFlight)
  fetchUpToMaxBytes() // 发送请求
  val numFetches = remoteRequests.size - fetchRequests.size
  logInfo("Started " + numFetches + " remote fetches in" + Utils.getUsedTimeMs
      (startTime))
  fetchLocalBlocks() // 获取本地Block
  logDebug("Got local blocks in " + Utils.getUsedTimeMs(startTime))
}
```

根据代码清单 8-84，ShuffleBlockFetcherIterator 的初始化过程如下。

1）给 TaskContextImpl 添加任务完成的监听器，以便于任务执行完成后调用 cleanup 方法进行一些清理工作。

2）使用 splitLocalRemoteBlocks 方法（见代码清单 8-85）划分从本地读取和需要远程读取的 Block 的请求。

3）将 FetchRequest 随机排序后存入 fetchRequests。

4）调用 fetchUpToMaxBytes 方法（见代码清单 8-86）发送请求。

5）调用 fetchLocalBlocks 方法（见代码清单 8-88）获取本地 Block。

2. 划分本地与远端 Block

ShuffleBlockFetcherIterator 的 splitLocalRemoteBlocks 方法（见代码清单 8-85）用于划分哪些 Block 从本地获取，哪些需要远程拉取，是获取中间计算结果的关键。

代码清单8-85 划分本地与远端Block

```
private[this] def splitLocalRemoteBlocks(): ArrayBuffer[FetchRequest] = {
  val targetRequestSize = math.max(maxBytesInFlight / 5, 1L)
  logDebug("maxBytesInFlight: " + maxBytesInFlight + ", targetRequestSize: " + target
      RequestSize)
  val remoteRequests = new ArrayBuffer[FetchRequest]
  var totalBlocks = 0
  for ((address, blockInfos) <- blocksByAddress) {
    totalBlocks += blockInfos.size
    if (address.executorId == blockManager.blockManagerId.executorId) { // 本地的
        Block
      localBlocks ++= blockInfos.filter(_._2 != 0).map(_._1)
      numBlocksToFetch += localBlocks.size // 将所有大小不为零的BlockId存入local
          Blocks
```

```
    } else { // 远端的Block
      val iterator = blockInfos.iterator
      var curRequestSize = 0L
      var curBlocks = new ArrayBuffer[(BlockId, Long)]
      while (iterator.hasNext) {
        val (blockId, size) = iterator.next()
        if (size > 0) {
          curBlocks += ((blockId, size)) // 将所有大小大于零的BlockId和size累加到
              curBlocks
          remoteBlocks += blockId // 将所有大小不为零的BlockId存入remoteBlocks
          numBlocksToFetch += 1
          curRequestSize += size // 增加当前请求要获取的Block的总大小
        } else if (size < 0) {
          throw new BlockException(blockId, "Negative block size " + size)
        }
        if (curRequestSize >= targetRequestSize) {
          remoteRequests += new FetchRequest(address, curBlocks) // 新建Fetch
              Request放入remoteRequests
          curBlocks = new ArrayBuffer[(BlockId, Long)]
          logDebug(s"Creating fetch request of $curRequestSize at $address")
          curRequestSize = 0
        }
      }
      if (curBlocks.nonEmpty) { // 对剩余的Block新建FetchRequest放入remoteRequests
        remoteRequests += new FetchRequest(address, curBlocks)
      }
    }
  }
  logInfo(s"Getting $numBlocksToFetch non-empty blocks out of $totalBlocks blocks")
  remoteRequests
}
```

这里为便于描述，先解释以下定义。

- targetRequestSize：每个远程请求的最大尺寸。targetRequestSize 等于 maxBytesInFlight 的 1/5 或者 1。
- totalBlocks：统计所有 Block 的总大小。
- curBlocks ：ArrayBuffer[(BlockId, Long)]，远程获取的累加缓存，用于保证每个远程请求的尺寸不超过 targetRequestSize 的限制。为什么要累加缓存？如果向一个机器节点频繁地请求字节数很小的 Block，那么势必造成网络拥塞并增加节点负担。将多个小数据量的请求合并为一个大的请求将避免这些问题，提高系统性能。
- curRequestSize ：当前累加到 curBlocks 中的所有 Block 的大小之和，用于保证每个远程请求的尺寸不超过 targetRequestSize 的限制。
- remoteRequests：new ArrayBuffer[FetchRequest]，缓存需要远程请求的 FetchRequest 对象。

明白了这些定义，我们一起来看看代码清单 8-85 所展示的 splitLocalRemoteBlocks 方法的处理逻辑。

1）遍历已经在 blocksByAddress 中缓存的按照 BlockManagerId 分组的 BlockId，如果

BlockManagerId 对应的 Executor 与当前 Executor 相同，则将 BlockManagerId 对应的所有大小不为零的 BlockId 存入 localBlocks；否则将 BlockManagerId 对应的所有大小大于零的 BlockId 和 size 累加到 curBlocks，将 BlockId 存入 remoteBlocks，curRequestSize 增加 size 的大小，每当 curRequestSize ≥ targetRequestSize，则新建 FetchRequest 放入 remoteRequests，并且为生成下一个 FetchRequest 做一些准备（如新建 curBlocks，curRequestSize 置为 0）。

2）遍历结束，curBlocks 中如果仍然有缓存的 (BlockId, Size)，新建 FetchRequest 放入 remoteRequests。此次请求不受 maxBytesInFlight 和 targetRequestSize 的影响。

为了使读者更容易理解划分本地与远端 Block 的执行过程，这里假设 maxBytesInFlight 等于 100，那么 targetRequestSize 等于 20。我用蓝色㊀代表本地的 Block，红色㊁和橙色㊂代表另外两个不同节点上的 Block，序列 blocksByAddress 中三个节点的顺序为蓝色、红色和橙色，那么划分本地与远端 Block 的执行过程如图 8-20 所示。

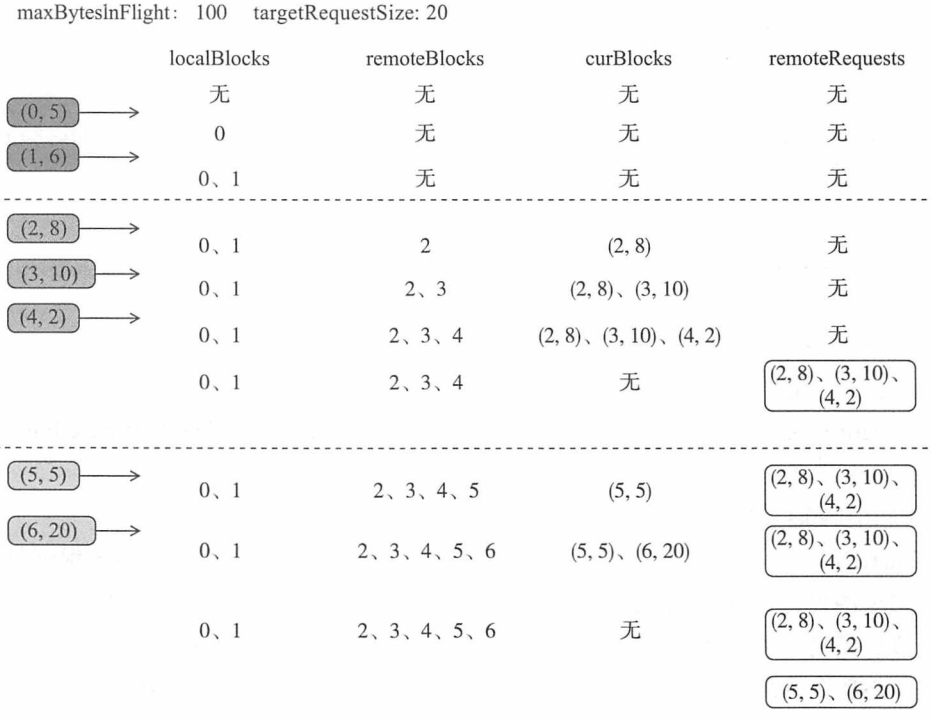


图 8-20 划分本地与远端 Block

这里对图 8-20 所示的内容进行简单的介绍。刚开始时，localBlocks、remoteBlocks、curBlocks、remoteRequests 中都没有数据。蓝、红、橙三个颜色的方块中的第一个数字代表

㊀ 虚线区隔的第一区域。
㊁ 虚线区隔的第二区域。
㊂ 虚线区隔的第三区域。

Block 的标识 BlockId（根据 6.2.1 节的内容，实际的 BlockId 并不是简单的数字，这里为了便于描述，才采用数字代替），而第二个数字代表 Block 的大小。对于本地的 Block，会将它们的 BlockId 放入 localBlocks，因此 localBlocks 中最终会保存 0 和 1 两个 BlockId。对于第一个远端节点上的三个 Block，会将它们的 BlockId 放入 remoteBlocks，因此 remoteBlocks 中最终会保存 2、3、4 三个 BlockId，同时会将 BlockId 及 Block 的大小放入 curBlocks 中。curBlocks 中三个 Block 的大小之和大于等于 targetRequestSize，所以将 curBlocks 及远端的地址（即 BlockManagerId）封装为 FetchRequest（以绿色边框的圆角矩形表示）放入 remoteRequests，之后新建 curBlocks。橙色 Block 的处理方式与红色类似，不再赘述。

3. 获取远端 Block

ShuffleBlockFetcherIterator 的 fetchUpToMaxBytes 方法（见代码清单 8-86）用于向远端发送请求，以获取 Block。

代码清单8-86 获取远端Block

```
private def fetchUpToMaxBytes(): Unit = {
  while (fetchRequests.nonEmpty &&
    (bytesInFlight == 0 ||
      (reqsInFlight + 1 <= maxReqsInFlight &&
        bytesInFlight + fetchRequests.front.size <= maxBytesInFlight))) {
    sendRequest(fetchRequests.dequeue())
  }
}
```

根据代码清单 8-86，fetchUpToMaxBytes 方法遍历 fetchRequests 中的所有 FetchRequest，远程请求 Block 中间结果。发送请求的前提是不超过 maxReqsInFlight 和 maxBytesInFlight 的限制。sendRequest 方法（见代码清单 8-87）将用于发送 FetchRequest 消息，以获取 Block。

代码清单8-87 发送FetchRequest消息

```
private[this] def sendRequest(req: FetchRequest) {
  logDebug("Sending request for %d blocks (%s) from %s".format(
    req.blocks.size, Utils.bytesToString(req.size), req.address.hostPort))
  bytesInFlight += req.size // 将请求的所有Block的大小累加到bytesInFlight
    reqsInFlight += 1 // 将reqsInFlight累加一

  val sizeMap = req.blocks.map { case (blockId, size) => (blockId.toString, size)
      }.toMap
  val remainingBlocks = new HashSet[String]() ++= sizeMap.keys
  val blockIds = req.blocks.map(_._1.toString)
  // 批量下载远端的Block
  val address = req.address
  shuffleClient.fetchBlocks(address.host, address.port, address.executorId, blockIds.
      toArray,
    new BlockFetchingListener {
      override def onBlockFetchSuccess(blockId: String, buf: ManagedBuffer): Unit = {
        ShuffleBlockFetcherIterator.this.synchronized {
```

```
            if (!isZombie) { // 结果封装为SuccessFetchResult放入results中
              buf.retain()
              remainingBlocks -= blockId
              results.put(new SuccessFetchResult(BlockId(blockId), address, sizeMap
                  (blockId), buf,
                remainingBlocks.isEmpty))
              logDebug("remainingBlocks: " + remainingBlocks)
            }
          }
          logTrace("Got remote block " + blockId + " after " + Utils.getUsedTimeMs
              (startTime))
        }

        override def onBlockFetchFailure(blockId: String, e: Throwable): Unit = {
          logError(s"Failed to get block(s) from ${req.address.host}:${req.address.
              port}", e)
          results.put(new FailureFetchResult(BlockId(blockId), address, e))
        }
      }
    )
  }
```

根据代码清单 8-87，sendRequest 方法的执行步骤如下。

1）将请求的所有 Block 的大小累加到 bytesInFlight。

2）将 reqsInFlight 累加 1。

3）调用 ShuffleClient 的 fetchBlocks 方法（请参阅 6.9.3 节）批量下载远端的 Block。

4）下载成功后将回调匿名 BlockFetchingListener 的 onBlockFetchSuccess 方法，将结果封装为 SuccessFetchResult 放入 results 中。

4. 获取本地 Block

ShuffleBlockFetcherIterator 的 fetchLocalBlocks 方法（见代码清单 8-88）用于获取本地 Block。

代码清单8-88　获取本地Block

```
private[this] def fetchLocalBlocks() {
  val iter = localBlocks.iterator
  while (iter.hasNext) {
    val blockId = iter.next() // 获取BlockId
    try { // 获取Block数据,创建SuccessFetchResult对象，并添加到results中
      val buf = blockManager.getBlockData(blockId)
      shuffleMetrics.incLocalBlocksFetched(1)
      shuffleMetrics.incLocalBytesRead(buf.size)
      buf.retain()
      results.put(new SuccessFetchResult(blockId, blockManager.blockManagerId, 0,
          buf, false))
    } catch {
      case e: Exception =>
        logError(s"Error occurred while fetching local blocks", e)
        results.put(new FailureFetchResult(blockId, blockManager.blockManagerId, e))
        return
    }
  }
}
```

根据代码清单 8-88，fetchLocalBlocks 方法将对 localBlocks 缓存的要在本地获取的 Block 进行迭代，每次迭代将执行以下操作。

1）获取 BlockId。

2）调用本地 BlockManager 的 getBlockData 方法（见代码清单 6-65）获取 Block。

3）进行一些 Shuffle 度量的更新。

4）创建 SuccessFetchResult 对象，并添加到 results 中。

5. 处理获取 Block 的结果信息

reduce 任务的上游 map 任务可能有多个，结合之前的分析，这些 map 任务输出的 Block（包括数据和索引）缓存在 ShuffleBlockFetcherIterator 的 results 队列中。ShuffleBlockFetcherIterator 继承了特质 scala.collection.Iterator，并实现了 hasNext 和 next 两个方法（见代码清单 8-89）。

代码清单8-89 ShuffleBlockFetcherIterator的实现

```
override def hasNext: Boolean = numBlocksProcessed < numBlocksToFetch
override def next(): (BlockId, InputStream) = {
  if (!hasNext) {
    throw new NoSuchElementException
  }

  numBlocksProcessed += 1
  val startFetchWait = System.currentTimeMillis()
  currentResult = results.take()
  val result = currentResult
  val stopFetchWait = System.currentTimeMillis()
  shuffleMetrics.incFetchWaitTime(stopFetchWait - startFetchWait)

  result match {
    case SuccessFetchResult(_, address, size, buf, isNetworkReqDone) =>
      if (address != blockManager.blockManagerId) {
        shuffleMetrics.incRemoteBytesRead(buf.size)
        shuffleMetrics.incRemoteBlocksFetched(1)
      }
      bytesInFlight -= size
      if (isNetworkReqDone) {
        reqsInFlight -= 1
        logDebug("Number of requests in flight " + reqsInFlight)
      }
    case _ =>
  }
  fetchUpToMaxBytes()
  result match {
    case FailureFetchResult(blockId, address, e) =>
      throwFetchFailedException(blockId, address, e)
    case SuccessFetchResult(blockId, address, _, buf, _) =>
      try {
        (result.blockId, new BufferReleasingInputStream(buf.createInputStream(), this))
      } catch {
        case NonFatal(t) =>
```

```
                throwFetchFailedException(blockId, address, t)
            }
        }
    }
```

根据代码清单 8-89，ShuffleBlockFetcherIterator 的 hasNext 是否为真，取决于 numBlocks Processed 是否小于 numBlocksToFetch 。next 方法每次迭代 ShuffleBlockFetcherIterator，会先从 results 队列中取出一个 FetchResult，然后根据 FetchResult 的类型匹配 FailureFetchResult 或 SuccessFetchResult。如果 Block 的获取结果为 SuccessFetchResult，那么返回 BlockId 与 BufferReleasingInputStream 的对偶。



注意 由于之前获取远端 Block 时，一小部分请求可能就达到了 maxBytesInFlight 的限制，所以很有可能会剩余很多请求没有发送。所以每此迭代 ShuffleBlockFetcher Iterator 的时候还有个附加动作用于发送剩余请求。

8.12.3 BlockStoreShuffleReader 详解

BlockStoreShuffleReader 用于 Shuffle 执行过程中，reduce 任务从其他节点的 Block 文件中读取由起始分区（startPartition）和结束分区（endPartition）指定范围内的数据。

在正式分析 BlockStoreShuffleReader 之前，我们先来了解其属性。

- handle：即 BaseShuffleHandle。
- startPartition：要读取的起始分区 ID（分区索引）。
- endPartition：要读取的结束分区 ID（分区索引）。
- context：即 TaskContext。
- serializerManager：即 SparkEnv 的子组件 SerializerManager。
- blockManager：即 SparkEnv 的子组件 BlockManager。
- mapOutputTracker：即 SparkEnv 的子组件 MapOutputTracker。
- dep：BaseShuffleHandle 的 dependency 属性，即 ShuffleDependency。

BlockStoreShuffleReader 中只有一个方法，那就是 read（见代码清单 8-90）。

代码清单8-90 BlockStoreShuffleReader的read方法

```
override def read(): Iterator[Product2[K, C]] = {
  // 伴随着对本地和远端的Block的获取
  val blockFetcherItr = new ShuffleBlockFetcherIterator(
    context,
    blockManager.shuffleClient,
    blockManager,
    mapOutputTracker.getMapSizesByExecutorId(handle.shuffleId, startPartition,
        endPartition),
    SparkEnv.get.conf.getSizeAsMb("spark.reducer.maxSizeInFlight", "48m") * 1024 *
        1024,
    SparkEnv.get.conf.getInt("spark.reducer.maxReqsInFlight", Int.MaxValue))
```

```
    val wrappedStreams = blockFetcherItr.map { case (blockId, inputStream) =>
      serializerManager.wrapStream(blockId, inputStream) // 对各个Block的输入流进行压缩
          和加密
    }
    val serializerInstance = dep.serializer.newInstance()
    val recordIter = wrappedStreams.flatMap { wrappedStream =>
      serializerInstance.deserializeStream(wrappedStream).asKeyValueIterator
    }
    val readMetrics = context.taskMetrics.createTempShuffleReadMetrics()
    val metricIter = CompletionIterator[(Any, Any), Iterator[(Any, Any)]](
      recordIter.map { record =>
        readMetrics.incRecordsRead(1)
        record
      },
      context.taskMetrics().mergeShuffleReadMetrics())
    val interruptibleIter = new InterruptibleIterator[(Any, Any)](context, metricIter)
    val aggregatedIter: Iterator[Product2[K, C]] = if (dep.aggregator.isDefined) {
      if (dep.mapSideCombine) { // 如果指定了聚合函数且允许在map端进行合并，在reduce端对数据
                                // 进行聚合
        val combinedKeyValuesIterator = interruptibleIter.asInstanceOf[Iterator[(K, C)]]
        dep.aggregator.get.combineCombinersByKey(combinedKeyValuesIterator, context)
      } else { // 如果指定了聚合函数，但不允许在map端进行合并，在reduce端对数据进行缓存
        val keyValuesIterator = interruptibleIter.asInstanceOf[Iterator[(K, Nothing)]]
        dep.aggregator.get.combineValuesByKey(keyValuesIterator, context)
      }
    } else { // 没有指定聚合函数，那么不作任何处理
      require(!dep.mapSideCombine, "Map-side combine without Aggregator specified!")
      interruptibleIter.asInstanceOf[Iterator[Product2[K, C]]]
    }

    dep.keyOrdering match {
      case Some(keyOrd: Ordering[K]) => // 如果指定了排序函数，则创建ExternalSorter
        val sorter =
          new ExternalSorter[K, C, C](context, ordering = Some(keyOrd), serializer
              = dep.serializer)
        sorter.insertAll(aggregatedIter) // 对数据进行缓存
        context.taskMetrics().incMemoryBytesSpilled(sorter.memoryBytesSpilled)
        context.taskMetrics().incDiskBytesSpilled(sorter.diskBytesSpilled)
        context.taskMetrics().incPeakExecutionMemory(sorter.peakMemoryUsedBytes)
        CompletionIterator[Product2[K, C], Iterator[Product2[K, C]]](sorter.iterator,
            sorter.stop())
      case None => // 如果没有指定排序函数，那么返回aggregatedIter
        aggregatedIter
    }
  }
```

根据代码清单 8-90，read 方法的执行步骤如下。

1）调用 MapOutputTracker 的 getMapSizesByExecutorId 方法（见代码清单 5-59）获取当前 reduce 任务所需的 map 任务中间输出数据的 BlockManager 的 BlockManagerId 及每个 Block 块的 BlockId 与大小。

2）创建 ShuffleBlockFetcherIterator。根据 8.12.2 节的介绍，ShuffleBlockFetcherIterator 在初始化时，ShuffleBlockFetcherIterator 就已经划分了本地与远端的 Block，并且获取了

本地和远端的 Block，最后将获取的数据封装为 SuccessFetchResult 或者 FailureFetchResult 后放入了 results 队列中。

3）根据配置对 ShuffleBlockFetcherIterator 中各个 Block 的输入流进行压缩和加密。

4）给每个流创建一个 key/value 的迭代器。

5）对任务的度量信息进行更新。

6）创建可中断的迭代器，即 InterruptibleIterator。这里使用 InterruptibleIterator，是为了能够支持任务尝试的取消操作。

7）如果指定了聚合函数且允许在 map 端进行合并，那么调用聚合器（Aggregator）的 combine CombinersByKey 方法（见代码清单 8-91）在 reduce 端对数据进行聚合，否则如果指定了聚合函数但不允许在 map 端进行合并，那么调用聚合器（Aggregator）的 combine ValuesByKey 方法（见代码清单 8-91）在 reduce 端对数据进行缓存，否则不作任何处理。

8）如果指定了排序函数，则创建 ExternalSorter，由于构造 ExternalSorter 时没有指定聚合函数和分区计算器，只指定了排序函数，因此调用 ExternalSorter 的 insertAll 方法时将使用 PartitionedPairBuffer 对数据进行缓存，最后封装并返回 CompletionIterator。如果没有指定排序函数，则返回第 6) 步得到的迭代器。

代码清单8-91 combineCombinersByKey与combineValuesByKey

```
def combineValuesByKey(
    iter: Iterator[_ <: Product2[K, V]],
    context: TaskContext): Iterator[(K, C)] = {
  val combiners = new ExternalAppendOnlyMap[K, V, C](createCombiner, mergeValue,
      mergeCombiners)
  combiners.insertAll(iter)
  updateMetrics(context, combiners)
  combiners.iterator
}

def combineCombinersByKey(
    iter: Iterator[_ <: Product2[K, C]],
    context: TaskContext): Iterator[(K, C)] = {
  val combiners = new ExternalAppendOnlyMap[K, C, C](identity, mergeCombiners,
      mergeCombiners)
  combiners.insertAll(iter)
  updateMetrics(context, combiners)
  combiners.iterator
}
```

根据代码清单 8-91，combineValuesByKey 和 combineCombinersByKey 方法都将调用 ExternalAppendOnlyMap 的 insertAll 方法（见代码清单 8-92）。

代码清单8-92 ExternalAppendOnlyMap的insertAll方法

```
def insertAll(entries: Iterator[Product2[K, V]]): Unit = {
  if (currentMap == null) {
    throw new IllegalStateException(
      "Cannot insert new elements into a map after calling iterator")
```

```
    }
    var curEntry: Product2[K, V] = null
    val update: (Boolean, C) => C = (hadVal, oldVal) => {
      if (hadVal) mergeValue(oldVal, curEntry._2) else createCombiner(curEntry._2)
    }
    while (entries.hasNext) {
      curEntry = entries.next()
      val estimatedSize = currentMap.estimateSize()
      if (estimatedSize > _peakMemoryUsedBytes) {
        _peakMemoryUsedBytes = estimatedSize
      }
      if (maybeSpill(currentMap, estimatedSize)) {
        currentMap = new SizeTrackingAppendOnlyMap[K, C]
      }
      currentMap.changeValue(curEntry._1, update)
      addElementsRead()
    }
  }
```

根据代码清单 8-92，ExternalAppendOnlyMap 的 insertAll 方法与 ExternalSorter 的 insertAll 方法的实现非常相似。ExternalAppendOnlyMap 与 ExternalSorter 有以下相同点。

- 都定义了用于更新和聚合的偏函数 update。
- 都使用了 AppendOnlyMap 的缓存聚合算法。
- 都进行了溢出判断和 AppendOnlyMap 底层 data 数据溢出到磁盘的操作。
- 都调用 addElementsRead 方法对已经读取的元素数量进行统计。

但二者也有很多不同之处，包括以下几点。

- ExternalAppendOnlyMap 使用的是 SizeTrackingAppendOnlyMap，而 ExternalSorter 使用的是 PartitionedAppendOnlyMap。
- ExternalSorter 会计算元素的分区 ID，而 ExternalAppendOnlyMap 不会。

8.12.4 SortShuffleManager 详解

SortShuffleManager 管理基于排序的 Shuffle——输入的记录按照目标分区 ID 排序，然后输出到一个单独的 map 输出文件中。reduce 为了读出 map 输出，需要获取 map 输出文件的连续内容。当 map 的输出数据太大已经不适合放在内存中时，排序后的输出子集将被溢出到文件中，这些磁盘上的文件将被合并生成最终的输出文件。

为便于分析 SortShuffleManager，应当从理解 SortShuffleManager 的属性开始。SortShuffleManager 包含的属性如下。

- numMapsForShuffle：Shuffle 的 ID 与为此 Shuffle 生成输出的 map 任务的数量之间的映射关系。
- shuffleBlockResolver：即 IndexShuffleBlockResolver。

SortShuffleManager 提供了很多方法，下面逐一介绍。

1. registerShuffle

registerShuffle 方法（见代码清单 8-93）用于根据条件创建不同的 ShuffleHandle 实例。

代码清单8-93 注册Shuffle

```
override def registerShuffle[K, V, C](
    shuffleId: Int,
    numMaps: Int,
    dependency: ShuffleDependency[K, V, C]): ShuffleHandle = {
  if (SortShuffleWriter.shouldBypassMergeSort(SparkEnv.get.conf, dependency)) {
    new BypassMergeSortShuffleHandle[K, V]( // 需要绕开合并及排序，则创建BypassMerge
        SortShuffleHandle
      shuffleId, numMaps, dependency.asInstanceOf[ShuffleDependency[K, V, V]])
  } else if (SortShuffleManager.canUseSerializedShuffle(dependency)) {
    new SerializedShuffleHandle[K, V]( // 如果可以使用序列化的Shuffle，则创建Serialized
        Shuffle Handle
      shuffleId, numMaps, dependency.asInstanceOf[ShuffleDependency[K, V, V]])
  } else {
    new BaseShuffleHandle(shuffleId, numMaps, dependency) // 其他情况，将创建Base
        ShuffleHandle
  }
}
```

根据代码清单 8-93，registerShuffle 方法的执行步骤如下。

1）调用 SortShuffleWriter 的伴生对象的 shouldBypassMergeSort 方法（见代码清单 8-78）后，发现如果需要绕开合并及排序，那么创建 BypassMergeSortShuffleHandle。

2）如果可以使用序列化的 Shuffle，那么创建 SerializedShuffleHandle。

3）其他情况，将创建 BaseShuffleHandle。

2. unregisterShuffle

unregisterShuffle 方法（见代码清单 8-94）用于根据指定的 shuffleId 删除此 Shuffle 过程的所有 map 任务的数据文件和索引文件。

代码清单8-94 unregisterShuffle方法的实现

```
override def unregisterShuffle(shuffleId: Int): Boolean = {
  Option(numMapsForShuffle.remove(shuffleId)).foreach { numMaps =>
    (0 until numMaps).foreach { mapId =>
      shuffleBlockResolver.removeDataByMap(shuffleId, mapId)
    }
  }
  true
}
```

根据代码清单 8-94，unregisterShuffle 是通过 IndexShuffleBlockResolver 的 removeDataByMap 方法（见代码清单 8-40）删除 map 任务产生的数据文件和索引文件的。

3. getWriter

getWriter 方法（见代码清单 8-95）用于根据 ShuffleHandle 获取 ShuffleWriter。

代码清单8-95 获取ShuffleWriter

```
override def getWriter[K, V](
    handle: ShuffleHandle,
    mapId: Int,
    context: TaskContext): ShuffleWriter[K, V] = {
  numMapsForShuffle.putIfAbsent( //将指定的shuffleId和Shuffle对应的map任务数注册到
      numMaps ForShuffle
    handle.shuffleId, handle.asInstanceOf[BaseShuffleHandle[_, _, _]].numMaps)
  val env = SparkEnv.get
  handle match { // 根据ShuffleHandle的具体类型，创建不同的ShuffleWriter
    case unsafeShuffleHandle: SerializedShuffleHandle[K @unchecked, V @unchecked] =>
      new UnsafeShuffleWriter(
        env.blockManager,
        shuffleBlockResolver.asInstanceOf[IndexShuffleBlockResolver],
        context.taskMemoryManager(),
        unsafeShuffleHandle,
        mapId,
        context,
        env.conf)
    case bypassMergeSortHandle: BypassMergeSortShuffleHandle[K @unchecked, V @
        unchecked] =>
      new BypassMergeSortShuffleWriter(
        env.blockManager,
        shuffleBlockResolver.asInstanceOf[IndexShuffleBlockResolver],
        bypassMergeSortHandle,
        mapId,
        context,
        env.conf)
    case other: BaseShuffleHandle[K @unchecked, V @unchecked, _] =>
      new SortShuffleWriter(shuffleBlockResolver, other, mapId, context)
  }
}
```

根据代码清单 8-95，getWriter 方法的执行步骤如下。

1）将指定 Shuffle 的 shuffleId 和 Shuffle 对应的 map 任务数注册到 numMapsForShuffle。

2）根据 ShuffleHandle 的具体类型，创建不同的 ShuffleWriter。

4. getReader

getReader 方法（见代码清单 8-96）用于获取对 map 任务输出的分区数据文件中从 startPartition 到 endPartition−1 范围内的数据进行读取的读取器（即 BlockStoreShuffleReader），供 reduce 任务使用。

代码清单8-96 获取BlockStoreShuffleReader

```
override def getReader[K, C](
    handle: ShuffleHandle,
    startPartition: Int,
    endPartition: Int,
    context: TaskContext): ShuffleReader[K, C] = {
  new BlockStoreShuffleReader(
    handle.asInstanceOf[BaseShuffleHandle[K, _, C]], startPartition, endPartition,
        context)
}
```

8.13 map 端与 reduce 端的 Shuffle 组合

经过对 map 端和 reduce 端执行代码的分析，读者对 Shuffle 的整个过程应该有了更深入的理解，并且能够对 Shuffle 进行性能调优。为了让代码分析的成果更加清晰，本书对 map 端和 reduce 端的 Shuffle 组合进行总结，共有以下几种。

1. map 端和 reduce 端都进行聚合

如果满足以下条件。

- ShuffleDependency 的 mapSideCombine 属性为 true（即允许在 map 端合并）。
- 指定了聚合函数。
- ShuffleDependency 不支持序列化。

那么 map 端与 reduce 端将以图 8-21 展示的方式组合。

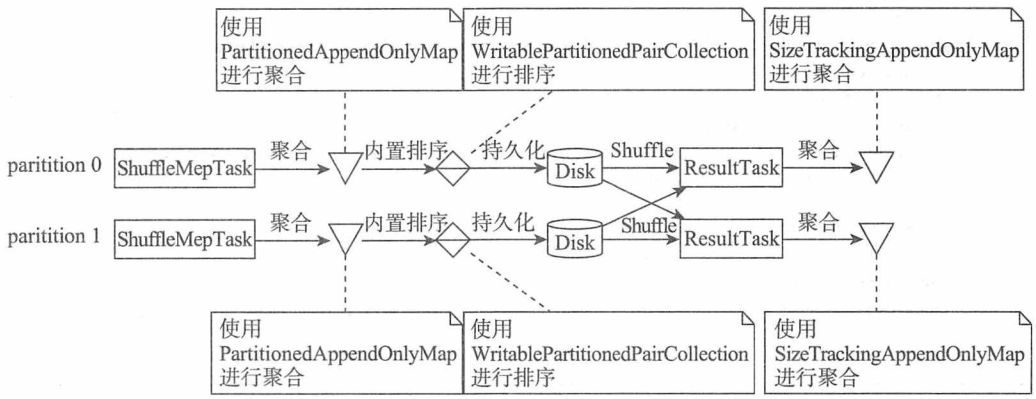


图 8-21 map 端和 reduce 端都进行聚合

如果指定了排序函数，还会在 reduce 端聚合后进行排序。

2. map 端缓存和 reduce 端聚合

如果满足以下条件。

- ShuffleDependency 的 mapSideCombine 属性为 false（即不允许在 map 端合并）。
- ShuffleDependency 的分区数大于 spark.shuffle.sort.bypassMergeThreshold 属性（默认为 200）指定的绕开合并和排序的阈值。
- ShuffleDependency 不支持序列化。
- 指定了聚合函数。

那么 map 端与 reduce 端将以图 8-22 展示的方式组合。

如果指定了排序函数，还会在 reduce 端聚合后进行排序。

3. map 端缓存和 reduce 端不聚合

如果满足以下条件。

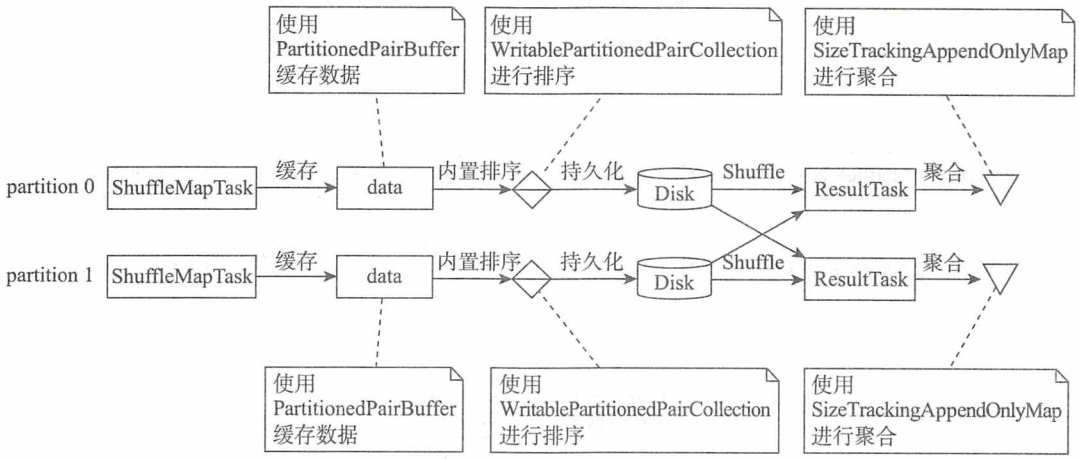


图 8-22 map 端缓存和 reduce 端聚合

- ShuffleDependency 的 mapSideCombine 属性为 false（即不允许在 map 端合并）。
- ShuffleDependency 的分区数大于 spark.shuffle.sort.bypassMergeThreshold 属性（默认为 200）指定的绕开合并的阈值。
- ShuffleDependency 不支持序列化。
- 没有指定聚合函数。

那么 map 端与 reduce 端将以图 8-23 展示的方式组合。

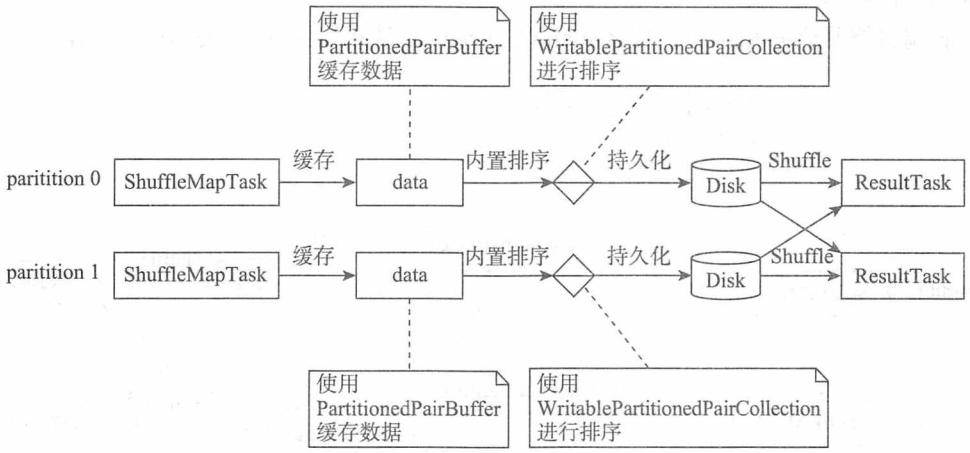


图 8-23 map 端缓存和 reduce 端不聚合

如果指定了排序函数，还会在 reduce 端进行排序。

此外，当 ShuffleDependency 支持序列化，其他三个条件不变时，map 端将使用 Unsafe ShuffleWriter。这种情况下的处理逻辑如图 8-24 所示。

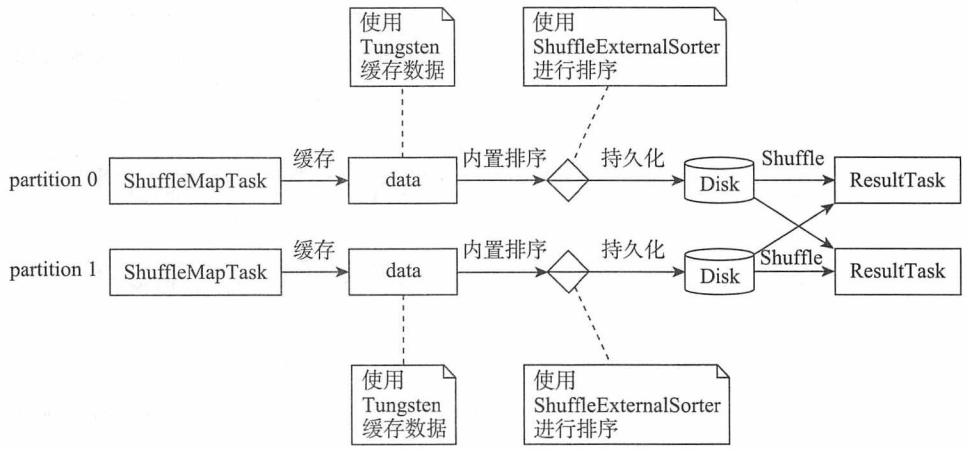


图 8-24　map 端使用 Tungsten 缓存和 reduce 端不聚合

如果指定了排序函数，还会在 reduce 端进行排序。

4. map 端绕开聚合、排序和 reduce 端不聚合

如果满足以下条件。

- ShuffleDependency 的 mapSideCombine 属性为 false（即不允许在 map 端合并）。
- ShuffleDependency 的分区数小于等于 spark.shuffle.sort.bypassMergeThreshold 属性（默认为 200）指定的绕开合并的阈值。
- 没有指定聚合函数。

map 端将绕开合并和排序，那么 map 端与 reduce 端将以图 8-25 展示的方式组合。

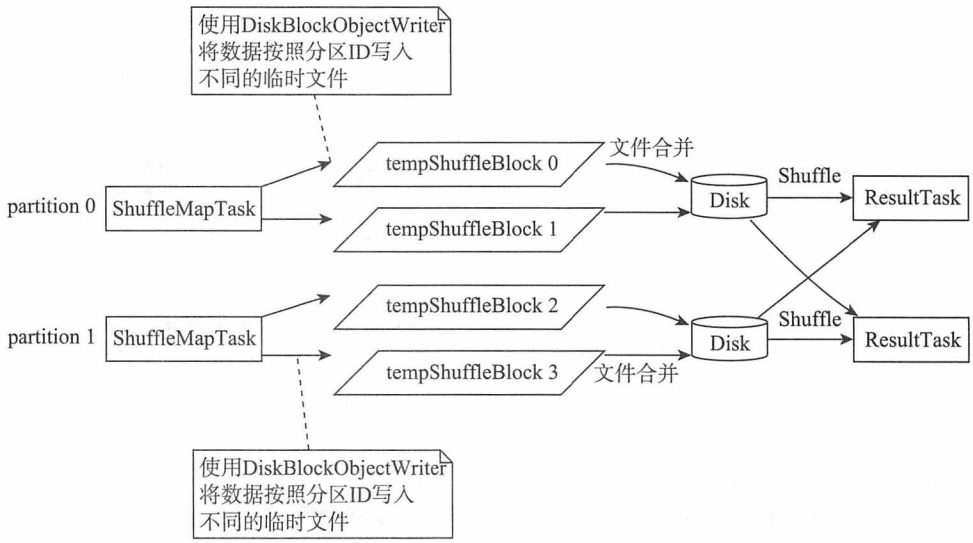


图 8-25　map 端临时 Shuffle 文件的合并与 reduce 端聚合

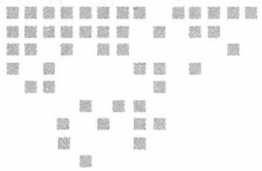
如果指定了聚合函数，则在 reduce 端会进行聚合。如果指定了排序函数，还会在 reduce 端聚合后进行排序。

8.14 小结

任何计算都离不开算法与程序，当开发人员写好了代码，就需要将它们提交到集群上作为任务执行。任务的执行离不开内存的使用，因此 Spark 提供了基于 JVM 堆内存的执行内存和 Tungsten 的内存。Tungsten 可以操纵的内存有 JVM 堆内存和操作系统内存两种。

为了提升程序执行效率，Spark 有时需要在 map 端对数据进行缓存、聚合、内置排序等操作。reduce 端为了提升效率，也可能需要对数据进行缓存、聚合、排序。

Shuffle 过程中，map 任务通过将多个分区的数据写入同一个文件，减轻了读写大量小文件给磁盘 I/O 效率带来的压力。reduce 任务通过对存储在同一个远端节点上的 Shuffle Block 进行积累，批量下载远端的 Block，节省了网络 I/O。



Chapter 9 第9章

部署模式

“事在四方，要在中央。圣人执要，四方来效。”

——《韩非子·物权》

本章导读

韩非子认为：“事情发生在天下四方，而要害却在中央”，所以他主张：“中央集权，决策权在圣人或皇帝手中，这样就可以掌管好整个国家、整个天下”。这种主张顺应了大一统的历史潮流，也迎合了秦始皇及后来封建帝王的需要。

Spark集群也是一个“天下”，在每个节点上时时刻刻都在发生着改变，为了对整个集群能够进行管理，也需要一个统领全局的“人”。集群管理器就是Spark集群中的“圣人”。针对不同的集群管理器，Spark提供了不同的部署模式对整个集群进行协调和管理。

之前的章节为了突出Spark各个组件的原理，凡是涉及部署模式的地方都以本地部署模式为主。无论哪种部署模式，大多数代码逻辑是通用的。但是各种部署模式在原理上存在诸多差异，例如，高可用性、容错与故障恢复、可扩展性等，本章将对这些差异进行分析。

1）Spark目前支持的部署模式如下。

- 本地部署模式：local、local[N]、local[*]、local[N, maxRetries]。此类模式主要用于代码调试和跟踪，不具备高可用性、可扩展性，所以不适用于生产环境。
- 本地集群部署模式：local-cluster[N, cores, memory]。此模式也用于代码调试和测试，是源码学习常用的模式，不具备高可用性、可扩展性，不适用于生产环境。

仅供非商业用途或交流学习使用

- Standalone 部署模式：spark://host:port。此模式具备高可用性并且支持分布式部署，所以可用于实际的生产环境。
- 第三方部署模式：YARN client、YARN cluster、mesos://、zk://、simr:// 等。

2）部署模式涉及很多角色。

- Application：用户使用 Spark API 开发的应用程序。
- Driver：应用驱动程序。有了 Driver，Application 才会被提交到 Spark 集群运行。Driver 可以选择在客户端运行，也可以选择向 Master 注册，然后由 Master 命令 Worker 启动 Driver。
- Master：Spark 的主控节点。在实际的生产环境中会有多个 Master，只有一个 Master 处于 active 状态，其余的 Master 处于 slave 状态。
- Worker：Spark 的工作节点，向 Master 汇报自身的资源、Executor 执行状态的改变，并接受 Master 的命令启动 Executor 或 Driver。
- Executor：Spark 的工作进程，由 Worker 监管，负责具体任务的执行。

9.1 心跳接收器 HeartbeatReceiver



在 4.4 节曾经介绍过 HeartbeatReceiver 的创建和注册，并对 HeartbeatReceiver 的作用进行了简单介绍。由于 HeartbeatReceiver 与部署模式息息相关，因此本节将深入分析 HeartbeatReceiver 的实现。

HeartbeatReceiver 运行在 Driver 上，用以接收各个 Executor 的心跳（HeartBeat）消息，对各个 Executor 的“生死”进行监控。HeartbeatReceiver 包含以下属性。

- sc：即 SparkContext。
- clock：类型为 org.apache.spark.util.Clock，Clock 实际是对 System.currentTimeMillis() 的一层封装，以便于获取系统时间。
- rpcEnv：即 SparkEnv 的子组件 RpcEnv。
- scheduler：即 TaskSchedulerImpl。
- executorLastSeen：用于维护 Executor 的身份标识与 HeartbeatReceiver 最后一次收到 Executor 的心跳（HeartBeat）消息的时间戳之间的映射关系。
- slaveTimeoutMs：Executor 节点上的 BlockManager 的超时时间（单位为 ms）。可通过 spark.storage.blockManagerSlaveTimeoutMs 属性配置，默认为 120000。
- executorTimeoutMs：Executor 的超时时间（单位为 ms）。可通过 spark.network.timeout 属性配置，默认为 120000。
- timeoutIntervalMs：超时的间隔（单位为 ms）。可通过 spark.storage.blockManagerTimeoutIntervalMs 属性配置，默认为 60000。
- checkTimeoutIntervalMs：检查超时的间隔（单位为 ms）。可通过 spark.network.time-

仅供非商业用途或交流学习使用

outInterval 属性配置，默认采用 timeoutIntervalMs 的值。

小贴士：spark.network.timeoutInterval 属性在配置时以 s 为单位，HeartbeatReceiver 会将秒转换为 ms。

- eventLoopThread：类型为 ScheduledThreadPoolExecutor，用于执行心跳接收器的超时检查任务，eventLoopThread 只包含一个线程，此线程以 heartbeat-receiver-event-loop-thread 作为名称。
- timeoutCheckingTask：向 eventLoopThread 提交执行超时检查的定时任务后返回的 ScheduledFuture。提交的定时任务的执行间隔为 checkTimeoutIntervalMs。
- killExecutorThread：以 Executors.newSingleThreadExecutor 方式创建的 ExecutorService，运行的单线程用于“杀死”（kill）Executor，此线程以 kill-executor-thread 作为名称。

了解了 HeartbeatReceiver 中的属性，现在来看看 HeartbeatReceiver 实现的方法。

1. 注册 Executor

HeartbeatReceiver 继承了 SparkListener，并实现了 onExecutorAdded 方法（见代码清单 9-1）。根据 3.3 节的内容，我们知道事件总线在接收到 SparkListenerExecutorAdded 消息后，将调用 HeartbeatReceiver 的 onExecutorAdded 方法，这样 HeartbeatReceiver 将监听到 Executor 的添加。



代码清单9-1 HeartbeatReceiver实现的onExecutorAdded方法

```
override def onExecutorAdded(executorAdded: SparkListenerExecutorAdded): Unit = {
  addExecutor(executorAdded.executorId)
}
```

根据代码清单 9-1，onExecutorAdded 方法从 SparkListenerExecutorAdded 事件中拿到 Executor 的身份标识后，将调用 addExecutor 方法。addExecutor 方法的实现如下。

```
def addExecutor(executorId: String): Option[Future[Boolean]] = {
  Option(self).map(_.ask[Boolean](ExecutorRegistered(executorId)))
}
```

根据上述代码，addExecutor 方法将向 HeartbeatReceiver 自己发送 ExecutorRegistered 消息。HeartbeatReceiver 继承了 ThreadSafeRpcEndpoint，并实现了 receiveAndReply 方法用以接收消息。HeartbeatReceiver 的 receiveAndReply 方法中处理 ExecutorRegistered 消息的代码如下。

```
case ExecutorRegistered(executorId) =>
  executorLastSeen(executorId) = clock.getTimeMillis()
  context.reply(true)
```

根据上述代码，HeartbeatReceiver 接收到 ExecutorRegistered 消息后，取出 Executor-

仅供非商业用途或交流学习使用

Registered 消息携带的要注册的 Executor 的身份标识，并把此 Executor 的身份标识与最后一次见面时间（即当前的时间戳）放入 executorLastSeen 中，最后向 HeartbeatReceiver 自己回复 true。

2. 移除 Executor

HeartbeatReceiver 继承了 SparkListener，并实现了 onExecutorRemoved 方法（见代码清单 9-2），这样 HeartbeatReceiver 将监听到 Executor 的移除。

代码清单9-2 HeartbeatReceiver实现的onExecutorRemoved方法

```
override def onExecutorRemoved(executorRemoved: SparkListenerExecutorRemoved): Unit = {
  removeExecutor(executorRemoved.executorId)
}
```

根据代码清单 9-2，onExecutorRemoved 方法从 SparkListenerExecutorRemoved 事件中拿到 Executor 的身份标识后，将调用 removeExecutor 方法。removeExecutor 方法的实现如下。

```
def removeExecutor(executorId: String): Option[Future[Boolean]] = {
  Option(self).map(_.ask[Boolean](ExecutorRemoved(executorId)))
}
```



根据上述代码，removeExecutor 方法将向 HeartbeatReceiver 自己发送 ExecutorRemoved 消息。HeartbeatReceiver 的 receiveAndReply 方法中处理 ExecutorRemoved 消息的代码如下。

```
case ExecutorRemoved(executorId) =>
  executorLastSeen.remove(executorId)
  context.reply(true)
```

根据上述代码，HeartbeatReceiver 接收到 ExecutorRemoved 消息后，取出 ExecutorRemoved 消息携带的要移除的 Executor 的身份标识，并从 executorLastSeen 中移除此 Executor 的缓存，最后向 HeartbeatReceiver 自己回复 true。

3. TaskSchedulerIsSet 消息

TaskSchedulerIsSet 消息表示 SparkContext 的 _taskScheduler 属性已经持有了 TaskScheduler 的引用。

在 4.5 节，我们曾经介绍过创建 TaskScheduler 和 DAGScheduler 的内容，根据代码清单 4-17，在创建了 TaskScheduler 后，SparkContext 的 _taskScheduler 属性将持有了 TaskScheduler 的引用，然后 SparkContext 将向 HeartbeatReceiver 发送 TaskSchedulerIsSet 消息。

HeartbeatReceiver 的 receiveAndReply 方法中处理 TaskSchedulerIsSet 消息的代码如下。

```
case TaskSchedulerIsSet =>
  scheduler = sc.taskScheduler
  context.reply(true)
```

根据上述代码，HeartbeatReceiver 接收到 TaskSchedulerIsSet 消息后，将获取 Spark-

仅供非商业用途或交流学习使用

Context 的 _taskScheduler 属性持有的 TaskScheduler 的引用，并由自身的 scheduler 属性保存，最后向 SparkContext 回复 true。

4. 检查超时的 Executor

HeartbeatReceiver 实现了 RpcEndpoint 的 onStart 方法，根据 5.3 节对 Dispatcher 的分析，我们知道当向 Dispatcher 注册 HeartbeatReceiver 时，与 HeartbeatReceiver 对应的 Inbox 将向自身的 messages 列表中放入 OnStart 消息，之后 Dispatcher 将处理 OnStart 消息，并调用 HeartbeatReceiver 的 onStart 方法。HeartbeatReceiver 的 onStart 方法的实现如代码清单 9-3 所示。

代码清单9-3 HeartbeatReceiver的onStart方法

```
override def onStart(): Unit = {
  timeoutCheckingTask = eventLoopThread.scheduleAtFixedRate(new Runnable {
    override def run(): Unit = Utils.tryLogNonFatalError {
      Option(self).foreach(_.ask[Boolean](ExpireDeadHosts))
    }
  }, 0, checkTimeoutIntervalMs, TimeUnit.MILLISECONDS)
}
```

根据代码清单 9-3，HeartbeatReceiver 的 onStart 方法将创建定时调度 timeoutCheckingTask。timeoutCheckingTask 按照 checkTimeoutIntervalMs 指定的间隔，向 HeartbeatReceiver 自身发送 ExpireDeadHosts 消息。HeartbeatReceiver 的 receiveAndReply 方法中处理 ExpireDeadHosts 消息的代码如下。

```
case ExpireDeadHosts =>
  expireDeadHosts()
  context.reply(true)
```

根据上述代码，HeartbeatReceiver 接收到 ExpireDeadHosts 消息后，将调用 expireDeadHosts 方法，然后向 HeartbeatReceiver 自己回复 true。expireDeadHosts 方法（见代码清单 9-4）用于检查超时的 Executor。

代码清单9-4 查超时的Executor

```
private def expireDeadHosts(): Unit = {
  logTrace("Checking for hosts with no recent heartbeats in HeartbeatReceiver.")
  val now = clock.getTimeMillis()
  for ((executorId, lastSeenMs) <- executorLastSeen) {
    if (now - lastSeenMs > executorTimeoutMs) {
      logWarning(s"Removing executor $executorId with no recent heartbeats: " +
        s"${now - lastSeenMs} ms exceeds timeout $executorTimeoutMs ms")
      scheduler.executorLost(executorId, SlaveLost("Executor heartbeat " +
        s"timed out after ${now - lastSeenMs} ms")) // 移除丢失的Executor
      killExecutorThread.submit(new Runnable {
        override def run(): Unit = Utils.tryLogNonFatalError {
          sc.killAndReplaceExecutor(executorId) // “杀死” Executor
        }
      })
```

仅供非商业用途或交流学习使用


```
        executorLastSeen.remove(executorId) // 移除对Executor的超时检查
      }
    }
  }
```

根据代码清单9-4，expireDeadHosts方法将遍历executorLastSeen中的每个Executor的身份标识对应的最后一次见面时间（lastSeenMs），并对当前时间（now）与lastSeenMs的差值大于executorTimeoutMs的Executor（这说明此Executor已经太长时间没有向HeartbeatReceiver发送HeartBeat了）作以下处理。

1）调用TaskSchedulerImpl的executorLost方法（见代码清单7-107）移除丢失的Executor，并对Executor上正在运行的Task重新分配资源后进行调度。

2）向单线程线程池killExecutorThread提交“杀死”Executor的任务。“杀死”Executor的任务通过调用SparkContext的killAndReplaceExecutor方法“杀死”Executor。

3）从executorLastSeen中移除此Executor。

小贴士：通过异步线程“杀死”（kill）Executor，是为了不阻塞执行expireDeadHosts方法的线程。

5. Executor的心跳

每个活跃的Executor都会向Driver上的HeartbeatReceiver发送HeartBeat消息（将在9.2.1节详细介绍），HeartbeatReceiver在接收到HeartBeat消息后，无论如何处理，都将向Executor回复HeartbeatResponse消息。HeartbeatResponse的实现如下。

```
private[spark] case class HeartbeatResponse(reregisterBlockManager: Boolean)
```

HeartbeatResponse消息携带的reregisterBlockManager表示是否要求Executor重新向BlockManagerMaster注册BlockManager。

HeartbeatReceiver的receiveAndReply方法中处理HeartBeat消息的代码如下。

```
case heartbeat @ Heartbeat(executorId, accumUpdates, blockManagerId) =>
  if (scheduler != null) {
    if (executorLastSeen.contains(executorId)) {
      executorLastSeen(executorId) = clock.getTimeMillis() // 更新此Executor的最
          后一次见面时间
      eventLoopThread.submit(new Runnable {
        override def run(): Unit = Utils.tryLogNonFatalError {
          val unknownExecutor = !scheduler.executorHeartbeatReceived(
            executorId, accumUpdates, blockManagerId)
          val response = HeartbeatResponse(reregisterBlockManager = unknown-
              Executor)
          context.reply(response) // 回复HeartbeatResponse消息
        }
      })
    } else {
      logDebug(s"Received heartbeat from unknown executor $executorId")
      context.reply(HeartbeatResponse(reregisterBlockManager = true))
```

仅供非商业用途或交流学习使用


```
    }
  } else {
    logWarning(s"Dropping $heartbeat because TaskScheduler is not ready yet")
    context.reply(HeartbeatResponse(reregisterBlockManager = true))
  }
```

根据上述代码，HeartbeatReceiver 接收到 HeartBeat 消息后，如果 TaskSchedulerImpl 为空，则向 Executor 发送 HeartbeatResponse 消息，此消息将会要求 BlockManager 重新注册。如果 TaskSchedulerImpl 不为空，那么将执行以下操作。

1）如果 executorLastSeen 中包含 HeartBeat 消息携带的 Executor 的身份标识，那么更新此 Executor 的最后一次见面时间，然后向 eventLoopThread 提交用于调用 TaskSchedulerImpl 的 executorHeartbeatReceived 方法的任务，最后向 Executor 回复 HeartbeatResponse 消息。TaskSchedulerImpl 的 executorHeartbeatReceived 方法用于更新正在处理的 Task 的度量信息，并且让 BlockManagerMaster 知道 BlockManager 仍然“活着”。

2）如果 executorLastSeen 中不包含 HeartBeat 消息携带的 Executor 的身份标识，那么向 Executor 发送 HeartbeatResponse 消息，此消息将会要求 BlockManager 重新注册。这种情况发生在刚刚从 executorLastSeen 中移除 Executor 及其最后见面时间后，却收到了 Executor 在被移除之前发送的 HeartBeat 消息。

经过对 HeartbeatReceiver 的实现分析，其工作原理可以用图 9-1 表示。

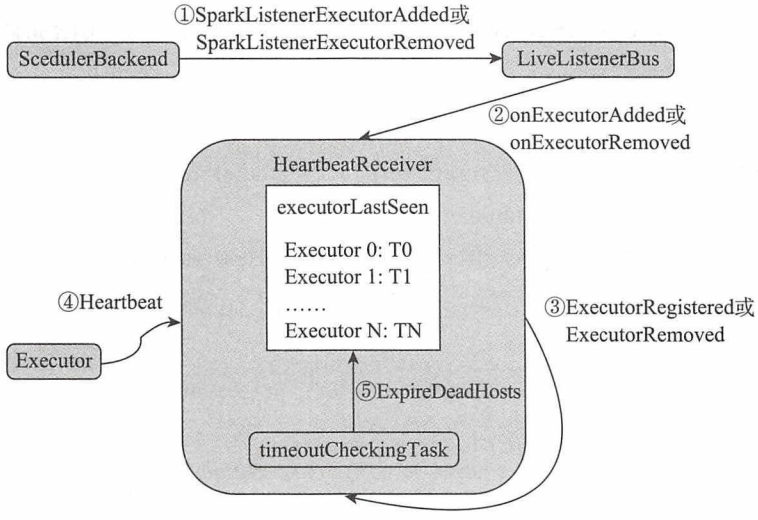


图 9-1 HeartbeatReceiver 的工作原理

这里对图 9-1 中的各个序号进行说明。

序号①：当 SchedulerBackend 已经创建或注册了 Executor 后，SchedulerBackend 将向 LiveListenerBus 投递 SparkListenerExecutorAdded 事件，当 Executor 与集群断开连接后，SchedulerBackend 将向 LiveListenerBus 投递 SparkListenerExecutorRemoved 事件。

仅供非商业用途或交流学习使用

序号②：由于 HeartbeatReceiver 也实现了 SparkListener，因此 LiveListenerBus 内部的异步线程在处理事件时，如果发现事件是 SparkListenerExecutorAdded 或 SparkListenerExecutorRemoved，将调用 HeartbeatReceiver 的 onExecutorAdded 或 onExecutorRemoved 方法。

序号③：当调用 HeartbeatReceiver 的 onExecutorAdded 或 onExecutorRemoved 方法时，HeartbeatReceiver 将向自己发送 ExecutorRegistered 消息或 ExecutorRemoved 消息。HeartbeatReceiver 接收到 ExecutorRegistered 消息，将把 Executor 的身份标识和当前系统时间作为对偶放入 executorLastSeen 缓存中。HeartbeatReceiver 接收到 ExecutorRemoved 消息，将把 Executor 的身份标识和对应的系统时间从 executorLastSeen 缓存中清除。

序号④：Executor 将不断向 HeartbeatReceiver 发送 Heartbeat 消息，HeartbeatReceiver 接收到 Heartbeat 消息后，将更新 executorLastSeen 缓存中 Executor 的身份标识对应的时间为系统当前时间。

序号⑤：HeartbeatReceiver 内部的检查 Executor 超时的定时任务，每隔一段时间将向 HeartbeatReceiver 自身发送 ExpireDeadHosts 消息，HeartbeatReceiver 接收到 ExpireDeadHosts 消息后，进行多项处理，移除丢失的 Executor、对 Executor 上正在运行的 Task 重新分配资源后进行调度、kill 丢失的 Executor、从 executorLastSeen 中移除此 Executor。



9.2 Executor 的实现分析

根据第 7 章对 LocalEndpoint 的 reviveOffers 方法（见代码清单 7-89）的介绍，我们知道最终会调用 Executor 的 launchTask 方法运行 Task。由于第 7 章主要介绍调度系统的内容，因此笔者没有分析 Executor 的实现，但现在已是继续深入的良好契机。

Executor 中包含以下属性。

- executorId：当前 Executor 的身份标识。
- executorHostname：当前 Executor 所在的主机名。
- env：即 SparkEnv。
- userClassPath：用户指定的类路径。可通过 spark.executor.extraClassPath 属性进行配置。如果有多个类路径，可以在配置时用英文逗号分隔。
- isLocal：是否是 local 部署模式。
- currentFiles：当前执行的 Task 所需要的文件。
- currentJars：当前执行的 Task 所需要的 Jar 包。
- EMPTY_BYTE_BUFFER：空的 java.nio.ByteBuffer，定义如下。

```
private val EMPTY_BYTE_BUFFER = ByteBuffer.wrap(new Array[Byte](0))
```

- conf：即 SparkConf。
- threadPool：使用 Executors.newCachedThreadPool 方式创建的 ThreadPoolExecutor，

仅供非商业用途或交流学习使用

用此线程池运行的线程将以 Executor task launch worker 为前缀。

- executorSource：类型为 ExecutorSource，是 Executor 的度量来源。
- taskReaperPool：使用 Executors.newCachedThreadPool 方式创建的 ThreadPoolExecutor，此线程池执行的线程用于监督 Task 的 kill 和取消。
- taskReaperForTask：用户缓存正在被 kill 的 Task 的身份标识与执行 kill 工作的任务收割者（TaskReaper）之间的映射关系。
- userClassPathFirst：是否首先从用户指定的类路径中加载类，然后再去 Spark 的 Jar 文件中加载。可通过 spark.executor.userClassPathFirst 属性配置，默认为 false。
- taskReaperEnabled：是否监控 kill 或中断 Task。可通过 spark.task.reaper.enabled 属性配置，默认为 false。
- urlClassLoader：Task 需要的类加载器。
- replClassLoader：当使用交互式解释器环境（Read-Evaluate-Print-Loop，简称 REPL）时，此类加载器用于加载 REPL 根据用户键入的代码定义的新类型。
- maxDirectResultSize：直接结果的最大大小。取 spark.task.maxDirectResultSize 属性（默认为 1L << 20，即 1 048 576）与 spark.rpc.message.maxSize 属性（默认为 128MB）之间的最小值。
- maxResultSize：结果的最大限制。此属性通过调用 Utils 工具类的 getMaxResultSize 方法获得，默认为 1GB。Task 运行的结果如果超过 maxResultSize，则会被删除。Task 运行的结果如果小于等于 maxResultSize 且大于 maxDirectResultSize，则会写入本地存储体系。Task 运行的结果如果小于等于 maxDirectResultSize，则会直接返回给 Driver。
- runningTasks：用于维护正在运行的 Task 的身份标识与 TaskRunner 之间的映射关系。
- heartbeater：只有一个线程的 ScheduledThreadPoolExecutor，此线程池运行的线程以 driver-heartbeater 作为名称。
- heartbeatReceiverRef：HeartbeatReceiver 的 RpcEndpointRef，通过调用 RpcEnv 的 setupEndpointRef 方法（见代码清单 5-42）获得。
- HEARTBEAT_MAX_FAILURES：心跳的最大失败次数。可通过 spark.executor.heartbeat.maxFailures 属性配置，默认为 60。
- heartbeatFailures：心跳失败数的计数器，初始值为 0。

在 Executor 的伴生对象中还定义了 ThreadLocal 属性 taskDeserializationProps。

```
val taskDeserializationProps: ThreadLocal[Properties] = new ThreadLocal[Properties]
```

9.2.1 Executor 的心跳报告

在初始化 Executor 的过程中，Executor 会调用自己的 startDriverHeartbeater 方法启动心跳报告的定时任务。startDriverHeartbeater 方法的实现如代码清单 9-5 所示。

代码清单9-5 启动心跳报告

```
private def startDriverHeartbeater(): Unit = {
  val intervalMs = conf.getTimeAsMs("spark.executor.heartbeatInterval", "10s")
  val initialDelay = intervalMs + (math.random * intervalMs).asInstanceOf[Int]

  val heartbeatTask = new Runnable() { // 向Driver报告心跳
    override def run(): Unit = Utils.logUncaughtExceptions(reportHeartBeat())
  }
  heartbeater.scheduleAtFixedRate(heartbeatTask, initialDelay, intervalMs, Time-
      Unit.MILLISECONDS)
}
```

根据代码清单 9-5，startDriverHeartbeater 方法的执行步骤如下。

1）获取发送心跳报告的时间间隔 intervalMs。可以看到，intervalMs 可通过 spark.executor.heartbeatInterval 属性配置，默认为 10s。

2）获取心跳定时器第一次执行的时间延迟 initialDelay。可以看到，initialDelay 是在 intervalMs 的基础上累加了随机数。

3）创建心跳任务 heartbeatTask。heartbeatTask 任务将通过 Utils 工具类的 logUncaughtExceptions 方法调用 reportHeartBeat 方法（见代码清单 9-6）报告心跳。

4）使 heartbeater 定时调度执行 heartbeatTask。

代码清单9-6 报告心跳

```
private def reportHeartBeat(): Unit = {
  val accumUpdates = new ArrayBuffer[(Long, Seq[AccumulatorV2[_, _]])]()
  val curGCTime = computeTotalGcTime()

  for (taskRunner <- runningTasks.values().asScala) {
    if (taskRunner.task != null) {
      taskRunner.task.metrics.mergeShuffleReadMetrics()
      taskRunner.task.metrics.setJvmGCTime(curGCTime - taskRunner.startGCTime)
      accumUpdates += ((taskRunner.taskId, taskRunner.task.metrics.accumulators()))
    }
  }

  val message = Heartbeat(executorId, accumUpdates.toArray, env.blockManager.
     blockManagerId)
  try { // 向Driver的HeartbeatReceiver发送Heartbeat消息
    val response = heartbeatReceiverRef.askWithRetry[HeartbeatResponse](
        message, RpcTimeout(conf, "spark.executor.heartbeatInterval", "10s"))
    if (response.reregisterBlockManager) {
      logInfo("Told to re-register on heartbeat")
      env.blockManager.reregister()
    }
    heartbeatFailures = 0
  } catch {
    case NonFatal(e) =>
      logWarning("Issue communicating with driver in heartbeater", e)
      heartbeatFailures += 1
      if (heartbeatFailures >= HEARTBEAT_MAX_FAILURES) {
        logError(s"Exit as unable to send heartbeats to driver " +
```

```
            s"more than $HEARTBEAT_MAX_FAILURES times")
          System.exit(ExecutorExitCode.HEARTBEAT_FAILURE)
        }
      }
    }
```

根据代码清单 9-6，reportHeartBeat 方法的执行步骤如下。

1）遍历 runningTasks 中正在运行的 Task，将每个 Task 的度量信息更新到数组缓冲 accumUpdates 中。

2）创建 Heartbeat 消息。

3）向 HeartbeatReceiver 发送 Heartbeat 消息，并接收 HeartbeatReceiver 的响应消息 HeartbeatResponse。

4）如果 HeartbeatResponse 的 reregisterBlockManager 属性为 true，此时调用 BlockManager 的 reregister 方法（见代码清单 6-60）向 BlockManagerMaster 重新注册 BlockManager。

5）将 heartbeatFailures 置为 0。

9.2.2 运行 Task

Executor 的 launchTask 方法用于运行 Task，其实现如代码清单 9-7 所示。

代码清单9-7 运行Task

```
def launchTask(
    context: ExecutorBackend,
    taskId: Long,
    attemptNumber: Int,
    taskName: String,
    serializedTask: ByteBuffer): Unit = {
  val tr = new TaskRunner(context, taskId = taskId, attemptNumber = attempt-
     Number, taskName,
    serializedTask)
  runningTasks.put(taskId, tr)
  threadPool.execute(tr)
}
```

根据代码清单 9-7，launchTask 方法的执行步骤如下。

1）创建 TaskRunner。

2）将 Task 的身份标识与 TaskRunner 的对应关系放入 runningTasks。

3）执行 TaskRunner。

TaskRunner 实现了 Runnable 接口，这样就可以作为线程要执行的任务。

TaskRunner 的构造器属性如下。

- execBackend：类型为 ExecutorBackend。根据 7.8 节对 SchedulerBackend 的介绍，我们知道 SchedulerBackend 有 CoarseGrainedSchedulerBackend 和 LocalSchedulerBackend 两个子类。由于 LocalSchedulerBackend 也继承了特质 ExecutorBackend，并实现了

其唯一的方法 statusUpdate，所以 local 部署模式下将 LocalSchedulerBackend 作为 ExecutorBackend。其他部署模式使用特质 ExecutorBackend 的另一个实现类 CoarseGrainedExecutorBackend，来跟 Coarse-GrainedSchedulerBackend 配合。

- taskId：Task 的身份标识。
- attemptNumber：任务尝试号。
- taskName：Task 的名称。
- serializedTask：序列化后的 Task。

TaskRunner 中的其他属性如下。

- threadName：线程名称。以 Executor task launch worker for task 为前缀，后跟 taskId。
- killed：Task 是否被 kill。
- threadId：线程的 ID。getThreadId 方法专门用于获取 threadId 的值。
- finished：任务是否已经完成。isFinished 方法专门用于获取 finished 的布尔值。
- startGCTime：任务尝试开始运行前，JVM 进程执行 GC 已花费的时间。
- task：要运行的 Task。通过对 Driver 传递过来的序列化的 Task 进行反序列化后获得。

TaskRunner 实现的 run 方法非常长，所以笔者将其实现分为以下六部分来讲解。

1. 运行准备

TaskRunner 实现的 run 方法中有关 Task 的运行准备的部分如下。

```
override def run(): Unit = {
  // 省略获取threadId、设置threadName的代码
  val taskMemoryManager = new TaskMemoryManager(env.memoryManager, taskId)
  // 省略次要代码
  Thread.currentThread.setContextClassLoader(replClassLoader)
  val ser = env.closureSerializer.newInstance()
  execBackend.statusUpdate(taskId, TaskState.RUNNING, EMPTY_BYTE_BUFFER) //更新
      Task为RUNNING
  var taskStart: Long = 0
  var taskStartCpu: Long = 0
  startGCTime = computeTotalGcTime() // 计算JVM进程执行GC已经花费的时间

  try {
    val (taskFiles, taskJars, taskProps, taskBytes) =
      Task.deserializeWithDependencies(serializedTask) // 对序列化的Task进行反序列化
    //Task所需的属性信息（taskProps）放入ThreadLocal
    Executor.taskDeserializationProps.set(taskProps)
    // 从taskFiles和taskJars中获取所需的资源依赖
    updateDependencies(taskFiles, taskJars)
    // 反序列化得到Task实例
    task = ser.deserialize[Task[Any]](taskBytes, Thread.currentThread.getContext-
        ClassLoader)
    task.localProperties = taskProps
    task.setTaskMemoryManager(taskMemoryManager) // 将TaskMemoryManager设置为
        Task的内存管理器
    // 省略当前TaskRunner的killed属性为true时，抛出TaskKilledException的代码
}
```

根据上述代码，Task 的运行准备工作如下。

1）创建 Task 尝试所需要的 TaskMemoryManager。

2）设置当前线程上下文的类加载器为 replClassLoader。

3）生成新的对闭包序列化的实例 ser。

4）调用 ExecutorBackend 的 statusUpdate 方法，将 Task 的状态更新为 RUNNING。可参考 LocalSchedulerBackend 实现的 statusUpdate 方法（见代码清单 7-93）或 CoarseGrainedExecutorBackend 的 statusUpdate 方法（见代码清单 9-92）。

5）调用 computeTotalGcTime 方法计算 JVM 进程执行 GC 已经花费的时间。

6）调用 Task 的 deserializeWithDependencies 方法对序列化的 Task 进行反序列化，得到任务所需的文件（taskFiles）、任务所需的 Jar 包（taskJars）、Task 所需的属性信息（taskProps）、任务本身（taskBytes）。

7）使用 Executor 的伴生对象的 taskDeserializationProps 将 taskProps 放入 ThreadLocal。

8）调用 updateDependencies 方法（见代码清单 9-8）从 taskFiles 和 taskJars 中获取所需的依赖。

9）将 Task 的 ByteBuffer（即 taskBytes）反序列化为 Task 实例。

10）将 taskProps 保存到 Task 的 localProperties 属性中。

11）将刚创建的 TaskMemoryManager 设置为 Task 的内存管理器。

代码清单9-8　更新依赖的文件和Jar包

```
private def updateDependencies(newFiles: HashMap[String, Long], newJars: Hash-
    Map[String, Long]) {
  lazy val hadoopConf = SparkHadoopUtil.get.newConfiguration(conf)
  synchronized {
    for ((name, timestamp) <- newFiles if currentFiles.getOrElse(name, -1L) <
       timestamp) {
      logInfo("Fetching " + name + " with timestamp " + timestamp)
      Utils.fetchFile(name, new File(SparkFiles.getRootDirectory()), conf,
        env.securityManager, hadoopConf, timestamp, useCache = !isLocal)
      currentFiles(name) = timestamp
    }
    for ((name, timestamp) <- newJars) {
      val localName = name.split("/").last
      val currentTimeStamp = currentJars.get(name)
        .orElse(currentJars.get(localName))
        .getOrElse(-1L)
      if (currentTimeStamp < timestamp) {
        logInfo("Fetching " + name + " with timestamp " + timestamp)
        Utils.fetchFile(name, new File(SparkFiles.getRootDirectory()), conf,
          env.securityManager, hadoopConf, timestamp, useCache = !isLocal)
        currentJars(name) = timestamp
        val url = new File(SparkFiles.getRootDirectory(), localName).toURI.toURL
        if (!urlClassLoader.getURLs().contains(url)) {
          logInfo("Adding " + url + " to class loader")
          urlClassLoader.addURL(url)
        }
```

```
        }
      }
    }
  }
```

根据代码清单 9-8，updateDependencies 方法首先下载 Task 所需的文件，并将文件名与时间戳的关系放入 currentFiles 缓存，然后下载 Task 所需的 Jar 包，并将 Jar 文件名与时间戳的关系放入 currentJars 缓存。Jar 包还将添加到 urlClassLoader 的加载路径中。下载文件都是通过 Utils 工具类的 fetchFile 方法，其具体实现请参阅附录 A。

2. 运行

TaskRunner 实现的 run 方法中有关运行 Task 的部分如代码清单 9-9 所示。

代码清单9-9 TaskRunner实现的run方法中有关运行Task的部分

```
val value = try {
  val res = task.run(
    taskAttemptId = taskId,
    attemptNumber = attemptNumber,
    metricsSystem = env.metricsSystem)
  threwException = false
  res
} finally {
// 省略finally里的代码
}
```

根据代码清单 9-9，TaskRunner 调用了 Task 的 run 方法（见代码清单 8-37），根据 8.5.2 节对 Task 的 run 方法的分析，我们知道 run 方法是一个模板方法，真正运行 Task 的是子类实现的 runTask 方法。

3. 资源回收

在上面的代码中，忽略了 finally 块中的代码，这些代码的实现如下。

```
val releasedLocks = env.blockManager.releaseAllLocksForTask(taskId) // 释放所有的
   Block锁
val freedMemory = taskMemoryManager.cleanUpAllAllocatedMemory() // 清空分配给任务尝
   试的内存

if (freedMemory > 0 && !threwException) {
  val errMsg = s"Managed memory leak detected; size = $freedMemory bytes, TID =
     $taskId"
  if (conf.getBoolean("spark.unsafe.exceptionOnMemoryLeak", false)) {
    throw new SparkException(errMsg)
  } else {
    logWarning(errMsg)
  }
}

if (releasedLocks.nonEmpty && !threwException) {
  val errMsg =
```

```
      s"${releasedLocks.size} block locks were not released by TID = $task-Id:\
          n" +
        releasedLocks.mkString("[", ", ", "]")
    if (conf.getBoolean("spark.storage.exceptionOnPinLeak", false)) {
      throw new SparkException(errMsg)
    } else {
      logWarning(errMsg)
    }
  }
```

根据上述代码，可以看到 finally 主要用于资源的回收，具体处理步骤如下。

1）调用 BlockManager 的 releaseAllLocksForTask 方法释放被当前任务尝试占用的所有 Block 的锁，并通知所有等待获取锁的线程。

2）调用 TaskMemoryManager 的 cleanUpAllAllocatedMemory 方法清空所有分配给当前任务尝试的 Page 和内存。

3）如果当前任务尝试有内存溢出或者存在 Block 的锁无法释放，则抛出异常或打印警告信息。

4. 执行结果序列化

下面的代码将对 Task 执行返回的结果 value 进行序列化处理。

```
val resultSer = env.serializer.newInstance()
val beforeSerialization = System.currentTimeMillis()
val valueBytes = resultSer.serialize(value)
val afterSerialization = System.currentTimeMillis()

// 省略度量更新相关代码
val accumUpdates = task.collectAccumulatorUpdates()
val directResult = new DirectTaskResult(valueBytes, accumUpdates)
val serializedDirectResult = ser.serialize(directResult)
val resultSize = serializedDirectResult.limit
```

5. 更新度量

在介绍执行结果的序列化时，忽略了其中对度量进行更新的代码，这些代码如下。

```
task.metrics.setExecutorDeserializeTime(
  (taskStart - deserializeStartTime) + task.executorDeserializeTime)
task.metrics.setExecutorDeserializeCpuTime(
  (taskStartCpu - deserializeStartCpuTime) + task.executorDeserialize-CpuTime)
task.metrics.setExecutorRunTime((taskFinish - taskStart) - task.executorDeseria-
    lizeTime)
task.metrics.setExecutorCpuTime(
  (taskFinishCpu - taskStartCpu) - task.executorDeserializeCpuTime)
task.metrics.setJvmGCTime(computeTotalGcTime() - startGCTime)
task.metrics.setResultSerializationTime(afterSerialization - beforeSerialization)
```

6. 执行结果发送

Task 运行的最后一步是将执行结果发送给 Driver 应用程序，如代码清单 9-10 所示。

代码清单9-10 执行结果的发送

```
val serializedResult: ByteBuffer = {
    if (maxResultSize > 0 && resultSize > maxResultSize) {
      // 将结果的大小序列化为serializedResult，并不会保存执行结果
      ser.serialize(new IndirectTaskResult[Any](TaskResultBlockId(taskId), result-
          Size))
    } else if (resultSize > maxDirectResultSize) {
      val blockId = TaskResultBlockId(taskId)
      env.blockManager.putBytes( // 将结果写入本地存储体系
        blockId,
        new ChunkedByteBuffer(serializedDirectResult.duplicate()),
        StorageLevel.MEMORY_AND_DISK_SER)
      // 将结果的大小序列化为serializedResult
      ser.serialize(new IndirectTaskResult[Any](blockId, resultSize))
    } else {
      logInfo(s"Finished $taskName (TID $taskId). $resultSize bytes result sent
          to driver")
      serializedDirectResult // 直接将结果序列化为serializedResult
    }
  }
  // 将Task的状态更新为FINISHED，并且将serializedResult发送给Driver
  execBackend.statusUpdate(taskId, TaskState.FINISHED, serializedResult)
```

根据代码清单 9-10，执行结果发送的处理逻辑如下。

1）如果 maxResultSize 大于 0 且任务尝试运行的结果超过 maxResultSize，那么只会将结果的大小序列化为 serializedResult，并且不会保存执行结果。

2）如果任务尝试运行的结果小于等于 maxResultSize 且大于 maxDirectResultSize，则会将结果写入本地存储体系，并将结果的大小序列化为 serializedResult。

3）如果任务尝试运行的结果小于等于 maxDirectResultSize，则会直接将结果序列化为 serializedResult。

4）调用 ExecutorBackend 的 statusUpdate 方法将 Task 的状态更新为 FINISHED，并且将 serializedResult 发送给 Driver。可参考 LocalSchedulerBackend 实现的 statusUpdate 方法（见代码清单 7-93）或 CoarseGrainedExecutorBackend 的 statusUpdate 方法（见代码清单 9-92）。

9.3 local 部署模式

本书在讲解各章内容时，主要以 local 模式为例。local 部署模式只有 Driver，没有 Master 和 Worker，执行任务的 Executor 与 Driver 在同一个 JVM 进程内。local 模式中使用的 ExecutorBackend 和 SchedulerBackend 的实现类都是 LocalSchedulerBackend。在第 7 章介绍调度系统时已经分析过 local 部署模式下的 LocalSchedulerBackend、LocalEndpoint 等组件的实现，本节将以图形的方式展现 local 部署模式的启动、local 部署模式下的任务提交与执行等流程。

local 部署模式的启动过程如图 9-2 所示。

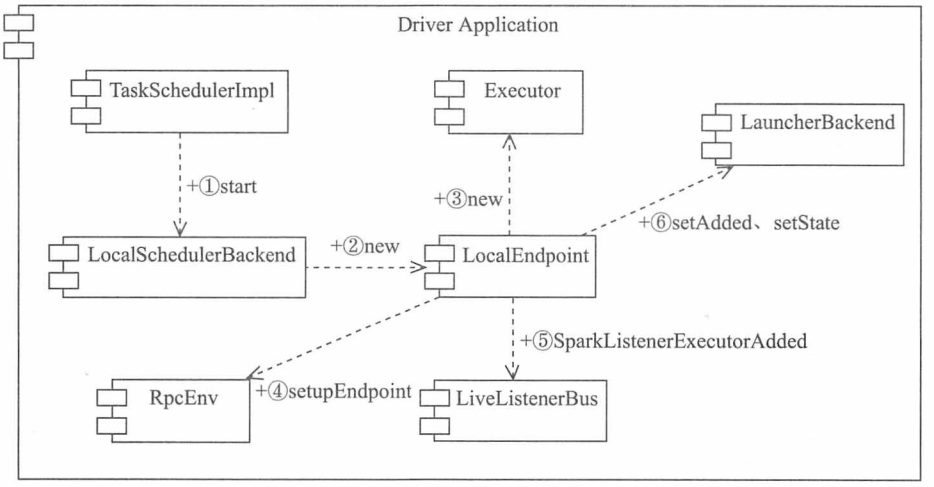


图 9-2　local 模式的启动过程

图 9-2 展示的执行过程已经在介绍 TaskSchedulerImpl 和 LocalSchedulerBackend 的启动时详细介绍过，所以不再赘述。

local 部署模式下的任务提交与执行过程如图 9-3 所示。

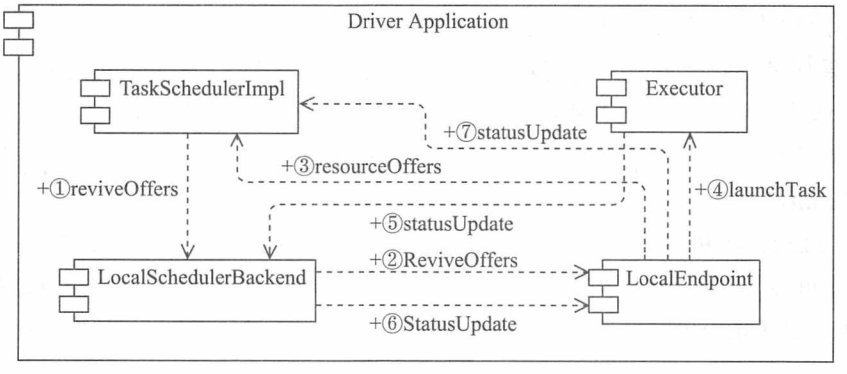


图 9-3　local 模式下的任务提交与执行

图 9-3 中 local 模式下的任务提交执行过程如下。

1）TaskSchedulerImpl 的 submitTasks 方法（见代码清单 7-99）在提交 Task 的最后会调用 LocalSchedulerBackend 的 reviveOffers 方法。

2）LocalSchedulerBackend 的 reviveOffers 方法（见代码清单 7-92）只是向 LocalEndpoint 发送 ReviveOffers 消息。

3）LocalEndpoint 收到 ReviveOffers 消息后（见代码清单 7-88），调用 TaskSchedulerImpl 的 resourceOffers 方法（见代码清单 7-101）申请资源，TaskSchedulerImpl 将根据任务申请的 CPU 核数、内存、本地化等条件为其分配资源。

4）任务获得资源后，调用 Executor 的 launchTask 方法（见代码清单 9-7）运行任务。

5）在任务运行过程（在 9.2.2 节已经详细介绍）中，Executor 中运行的 TaskRunner 通过调用 LocalSchedulerBackend 的 statusUpdate 方法更新 Task 的状态。

6）LocalSchedulerBackend 的 statusUpdate 方法将向 LocalEndpoint 发送 StatusUpdate 消息。

7）LocalEndpoint 接收到 StatusUpdate 消息，将调用 TaskSchedulerImpl 的 statusUpdate 方法（见代码清单 7-103）更新任务的状态。

9.4 持久化引擎 PersistenceEngine

PersistenceEngine 用于当 Master 发生故障后，通过领导选举选择其他 Master 接替整个集群的管理工作时，能够使得新激活的 Master 有能力从故障中恢复整个集群的状态信息，进而恢复对集群资源的管理和分配。抽象类 PersistenceEngine 定义了对 Master 必需的任何状态信息进行持久化的接口规范。实现 PersistenceEngine 必须满足以下机制。

- 在完成新的 Application 的注册之前，addApplication 方法必须被调用。
- 在完成新的 Worker 的注册之前，addWorker 方法必须被调用。
- removeApplication 方法和 removeWorker 方法可以在任何时候调用。

在以上机制的保证下，整个集群的 Worker、Driver 和 Application 的信息都被持久化，集群因此可以在领导选举和主 Master 的切换后，对集群状态进行恢复。

抽象类 PersistenceEngine 的定义如代码清单 9-11 所示。

代码清单9-11 PersistenceEngine的定义

```
@DeveloperApi
abstract class PersistenceEngine {
  def persist(name: String, obj: Object): Unit
  def unpersist(name: String): Unit
  def read[T: ClassTag](prefix: String): Seq[T]
  final def addApplication(app: ApplicationInfo): Unit = {
    persist("app_" + app.id, app)
  }
  final def removeApplication(app: ApplicationInfo): Unit = {
    unpersist("app_" + app.id)
  }
  final def addWorker(worker: WorkerInfo): Unit = {
    persist("worker_" + worker.id, worker)
  }
  final def removeWorker(worker: WorkerInfo): Unit = {
    unpersist("worker_" + worker.id)
  }
  final def addDriver(driver: DriverInfo): Unit = {
    persist("driver_" + driver.id, driver)
  }
```

```
  final def removeDriver(driver: DriverInfo): Unit = {
    unpersist("driver_" + driver.id)
  }
  final def readPersistedData(
      rpcEnv: RpcEnv): (Seq[ApplicationInfo], Seq[DriverInfo], Seq[WorkerInfo]) = {
    rpcEnv.deserialize { () =>
      (read[ApplicationInfo]("app_"), read[DriverInfo]("driver_"), read[Worker-
          Info]("worker_"))
    }
  }
  def close() {}
}
```

根据代码清单 9-11，抽象类 PersistenceEngine 定义的方法分别如下。

- persist：对对象进行序列化和持久化的抽象方法，具体的实现依赖于底层使用的存储。
- unpersist：对对象进行非持久化（从存储中删除）的抽象方法，具体的实现依赖于底层使用的存储。
- read：读取匹配给定前缀的所有对象。
- addApplication：对应用的信息（ApplicationInfo）进行持久化的模板方法，其依赖于 persist 的具体实现。
- removeApplication：对应用的信息（ApplicationInfo）进行非持久化的模板方法，其依赖于 unpersist 的具体实现。
- addWorker：对 Worker 的信息（WorkerInfo）进行持久化的模板方法，其依赖于 persist 的具体实现。
- removeWorker：对 Worker 的信息（WorkerInfo）进行非持久化的模板方法，其依赖于 unpersist 的具体实现。
- addDriver：对 Driver 的信息（DriverInfo）进行持久化的模板方法，其依赖于 persist 的具体实现。
- removeDriver：对 Driver 的信息（DriverInfo）进行非持久化的模板方法，其依赖于 unpersist 的具体实现。
- readPersistedData：读取并反序列化所有持久化的数据。
- close：用于关闭 PersistenceEngine。

抽象类 PersistenceEngine 有四个实现子类，如图 9-4 所示。

从图 9-4 中可以看到，PersistenceEngine 一共有四种实现，下面对它们作简单介绍。BlackHolePersistenceEngine 正如“黑洞”一样，其实现都是空实现。

```
private[master] class BlackHolePersistenceEngine extends PersistenceEngine {
  override def persist(name: String, obj: Object): Unit = {}
  override def unpersist(name: String): Unit = {}
  override def read[T: ClassTag](name: String): Seq[T] = Nil
}
```

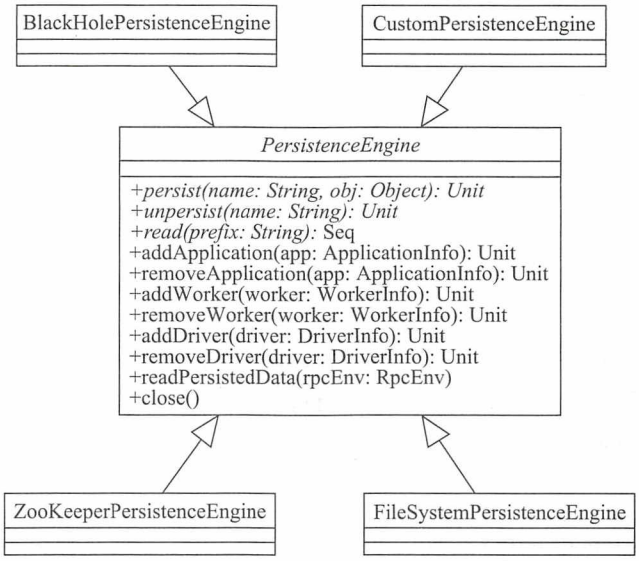


图 9-4 PersistenceEngine 的继承体系

CustomPersistenceEngine 是用于单元测试的实现。FileSystemPersistenceEngine 和 ZooKeeperPersistenceEngine 是真正可用于生产环境的实现类，因此本节将主要对后两者进行分析。

9.4.1 基于文件系统的持久化引擎

FileSystemPersistenceEngine 是基于文件系统的持久化引擎。对于 ApplicationInfo、WorkerInfo 及 DriverInfo，FileSystemPersistenceEngine 会将它们的数据存储到磁盘上的单个文件夹中，当要移除它们时，这些磁盘文件将被删除。由于不同的 Master 往往不在同一个机器节点上，因此在使用 FileSystemPersistenceEngine 时，底层的文件系统应该是分布式的。

FileSystemPersistenceEngine 一共有两个属性，分别如下。

- dir：持久化的根目录。
- serializer：持久化时使用的序列化器。

了解了 FileSystemPersistenceEngine 的属性，下面将逐一介绍它实现的方法。

1. persist

FileSystemPersistenceEngine 实现的 persist 方法如代码清单 9-12 所示。

代码清单9-12 FileSystemPersistenceEngine的persist方法

```
override def persist(name: String, obj: Object): Unit = {
  serializeIntoFile(new File(dir + File.separator + name), obj)
}
private def serializeIntoFile(file: File, value: AnyRef) {
  val created = file.createNewFile()
```

```
    if (!created) { throw new IllegalStateException("Could not create file: " + file) }
    val fileOut = new FileOutputStream(file)
    var out: SerializationStream = null
    Utils.tryWithSafeFinally {
      out = serializer.newInstance().serializeStream(fileOut)
      out.writeObject(value)
    } {
      fileOut.close()
      if (out != null) {
        out.close()
      }
    }
  }
```

根据代码清单 9-12，persist 方法实际是通过创建文件，打开文件输出流，并对数据进行序列化后写入磁盘的。

2. unpersist

FileSystemPersistenceEngine 实现的 unpersist 方法实际是通过删除对应的文件实现的，如代码清单 9-13 所示。

代码清单9-13 FileSystemPersistenceEngine的unpersist方法

```
override def unpersist(name: String): Unit = {
  val f = new File(dir + File.separator + name)
  if (!f.delete()) {
    logWarning(s"Error deleting ${f.getPath()}")
  }
}
```

3. read

FileSystemPersistenceEngine 实现的 read 方法（见代码清单 9-14）实际是通过过滤出持久化根目录下，文件名匹配前缀的所有文件，并对每个文件的数据反序列化实现的。

代码清单9-14 FileSystemPersistenceEngine的read方法

```
override def read[T: ClassTag](prefix: String): Seq[T] = {
  val files = new File(dir).listFiles().filter(_.getName.startsWith(prefix))
  files.map(deserializeFromFile[T])
}

private def deserializeFromFile[T](file: File)(implicit m: ClassTag[T]): T = {
  val fileIn = new FileInputStream(file)
  var in: DeserializationStream = null
  try {
    in = serializer.newInstance().deserializeStream(fileIn)
    in.readObject[T]()
  } finally {
    fileIn.close()
    if (in != null) {
      in.close()
    }
```

```
  }
}
```

9.4.2 基于 ZooKeeper 的持久化引擎

ZooKeeperPersistenceEngine 是基于 ZooKeeper 的持久化引擎。对于 ApplicationInfo、WorkerInfo 及 DriverInfo，ZooKeeperPersistenceEngine 会将它们的数据存储到 ZooKeeper 的不同节点（也称为 Znode）中，当要移除它们时，这些节点将被删除。

ZooKeeperPersistenceEngine 有以下属性。

- conf：即 SparkConf。
- serializer：持久化时使用的序列化器。
- WORKING_DIR：ZooKeeperPersistenceEngine 在 ZooKeeper 上的工作目录，是 Spark 基于 ZooKeeper 进行热备的根节点（可通过 spark.deploy.ZooKeeper.dir 属性配置，默认为 spark）的子节点 master_status。
- zk：连接 ZooKeeper 的客户端，类型为 CuratorFramework。

小贴士： Curator 是 Netflix 公司开源的一个 ZooKeeper 客户端，与 ZooKeeper 提供的原生客户端相比，Curator 的抽象层次更高，其核心目标是帮助工程师管理 ZooKeeper 的相关操作，简化 ZooKeeper 客户端的开发量。Curator 现已提升为 Apache 的顶级项目。CuratorFramework 就是 Curator 提供的 API。

了解了 ZooKeeperPersistenceEngine 的属性，下面将逐一介绍它实现的方法。

1. persist

ZooKeeperPersistenceEngine 实现的 persist 方法如代码清单 9-15 所示。

代码清单9-15 ZooKeeperPersistenceEngine的persist方法

```
override def persist(name: String, obj: Object): Unit = {
  serializeIntoFile(WORKING_DIR + "/" + name, obj)
}
private def serializeIntoFile(path: String, value: AnyRef) {
  val serialized = serializer.newInstance().serialize(value)
  val bytes = new Array[Byte](serialized.remaining())
  serialized.get(bytes)
  zk.create().withMode(CreateMode.PERSISTENT).forPath(path, bytes)
}
```

根据代码清单 9-15，persist 方法实际是通过对数据进行序列化，然后将序列化的数据保存到 ZooKeeper 的 master_status 节点下创建的持久化子节点中实现的。

小贴士： ZooKeeper 的节点的创建模式（CreateMode）共有四种，分别是持久化（PERSISTENT）、持久化与顺序化（PERSISTENT_SEQUENTIAL）、临时（EPHEMERAL）、

临时与顺序化（EPHEMERAL_SEQUENTIAL）。对于顺序化的节点，ZooKeeper 会为其创建自增长的序号。对于临时节点，ZooKeeper 会在创建它的客户端退出后自动删除。对于持久化的节点，ZooKeeper 不会在创建它的客户端退出后自动删除，需要客户端主动删除。

2. unpersist

ZooKeeperPersistenceEngine 实现的 unpersist 方法实际是通过删除对应的 ZooKeeper 节点实现的，如代码清单 9-16 所示。

代码清单9-16　ZooKeeperPersistenceEngine的unpersist方法

```
override def unpersist(name: String): Unit = {
  zk.delete().forPath(WORKING_DIR + "/" + name)
}
```

3. read

ZooKeeperPersistenceEngine 实现的 read 方法实际是通过过滤出 ZooKeeper 的 master_status 节点下文件名匹配前缀的所有子节点，并对每个子节点的数据反序列化实现的，如代码清单 9-17 所示。

代码清单9-17　ZooKeeperPersistenceEngine的read方法

```
override def read[T: ClassTag](prefix: String): Seq[T] = {
  zk.getChildren.forPath(WORKING_DIR).asScala
    .filter(_.startsWith(prefix)).flatMap(deserializeFromFile[T])
}
private def deserializeFromFile[T](filename: String)(implicit m: ClassTag[T]):
    Option[T] = {
  val fileData = zk.getData().forPath(WORKING_DIR + "/" + filename)
  try {
    Some(serializer.newInstance().deserialize[T](ByteBuffer.wrap(fileData)))
  } catch {
    case e: Exception =>
      logWarning("Exception while reading persisted file, deleting", e)
      zk.delete().forPath(WORKING_DIR + "/" + filename)
      None
  }
}
```

4. close

ZooKeeperPersistenceEngine 实现的 close 方法实际只是关闭了连接 ZooKeeper 的客户端，代码如下。

```
override def close() {
  zk.close()
}
```

9.5　领导选举代理

领导选举机制（Leader Election）可以保证集群虽然存在多个 Master，但是只有一

个 Master 处于激活（Active）状态，其他的 Master 处于支持（Standby）状态。当 Active 状态的 Master 出现故障时，会选举出一个 Standby 状态的 Master 作为新的 Active 状态的 Master。由于整个集群的 Worker，Driver 和 Application 的信息都已经通过持久化引擎持久化，因此切换 Master 时只会影响新任务的提交，对于正在运行中的任务没有任何影响。

特质 LeaderElectionAgent 定义了对当前的 Master 进行跟踪和领导选举代理的通用接口，其定义如下。

```
@DeveloperApi
trait LeaderElectionAgent {
  val masterInstance: LeaderElectable
  def stop() {} // to avoid noops in implementations.
}
```

上述代码中的 masterInstance 属性的类型是 LeaderElectable，特质 LeaderElectable 的定义如下。

```
@DeveloperApi
trait LeaderElectable {
  def electedLeader(): Unit
  def revokedLeadership(): Unit
}
```

LeaderElectable 定义了两个方法。

- electedLeader：被选举为领导。
- revokedLeadership：撤销领导关系。

LeaderElectable 目前只有 Master 这一个实现类。

特质 LeaderElectionAgent 的继承体系如图 9-5 所示。

从图 9-5 中，可以看到 LeaderElectionAgent 一共有三种实现。由于 CustomLeaderElectionAgent 用于单元测试，所以我们不对其过多介绍。其余两种实现可用于生产环境，本节将逐一进行介绍。

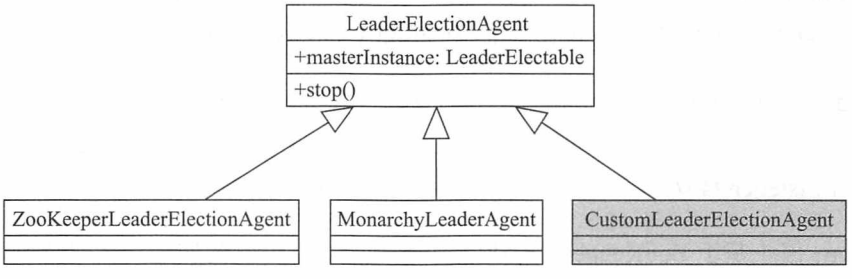


图 9-5 LeaderElectionAgent 的继承体系

1. MonarchyLeaderAgent 详解

MonarchyLeaderAgent 的实现非常简单，如代码清单 9-18 所示。

根据代码清单 9-18，可以看到 MonarchyLeaderAgent 在构造时会调用 masterInstance 的 electedLeader 方法选举领导。本章将在 9.6 节分析 Master 实现的 electedLeader 方法。

代码清单9-18 MonarchyLeaderAgent的实现

```
private[spark] class MonarchyLeaderAgent(val masterInstance: LeaderElectable)
  extends LeaderElectionAgent {
  masterInstance.electedLeader()
}
```

2. ZooKeeperLeaderElectionAgent 详解

ZooKeeperLeaderElectionAgent 是借助 ZooKeeper 实现的领导选举代理。ZooKeeper-LeaderElectionAgent 中的属性如下。

1）WORKING_DIR：ZooKeeperLeaderElectionAgent 在 ZooKeeper 上的工作目录，是 Spark 基于 ZooKeeper 进行热备的根节点（可通过 spark.deploy.ZooKeeper.dir 属性配置，默认为 spark）的子节点 leader_election。

2）zk：连接 ZooKeeper 的客户端，类型为 CuratorFramework。

3）leaderLatch：使用 ZooKeeper 进行领导选举的客户端，类型为 LeaderLatch。

4）status：领导选举的状态，包括有领导（LEADER）和无领导（NOT_LEADER）。

了解了 ZooKeeperLeaderElectionAgent 的属性，下面将逐一介绍它实现的方法。

5）start：启动基于 ZooKeeper 的领导选举代理。start 的实现如代码清单 9-19 所示。

代码清单9-19 启动ZooKeeperLeaderElectionAgent

```
private def start() {
  logInfo("Starting ZooKeeper LeaderElection agent")
  zk = SparkCuratorUtil.newClient(conf)
  leaderLatch = new LeaderLatch(zk, WORKING_DIR)
  leaderLatch.addListener(this)
  leaderLatch.start()
}
```

由于 ZooKeeperLeaderElectionAgent 实现了 LeaderLatchListener，因此发生领导选举后，LeaderLatch 会回调 ZooKeeperLeaderElectionAgent 的 isLeader 或 notLeader 方法。

小贴士：LeaderLatch 是 Curator 提供的进行主节点选举的 API。为了在 Curator 客户端获得或者失去管理权时进行回调处理，需要注册一个 LeaderLatchListener 接口的实现类。LeaderLatchListener 接口中定义了 isLeader 和 notLeader 两个方法，需要实现类实现。

6）stop：停止基于 ZooKeeper 的领导选举代理。stop 方法的实现如代码清单 9-20 所示。

代码清单9-20 停止ZooKeeperLeaderElectionAgent

```
override def stop() {
  leaderLatch.close()
  zk.close()
```

```
}
```

7）updateLeadershipStatus：更新领导关系状态。updateLeadershipStatus 方法的实现如代码清单 9-21 所示。

代码清单9-21 更新领导关系状态

```
private def updateLeadershipStatus(isLeader: Boolean) {
  if (isLeader && status == LeadershipStatus.NOT_LEADER) { // 选举为领导
    status = LeadershipStatus.LEADER
    masterInstance.electedLeader()
  } else if (!isLeader && status == LeadershipStatus.LEADER) {// 撤销领导职务
    status = LeadershipStatus.NOT_LEADER
    masterInstance.revokedLeadership()
  }
}
```

根据代码清单 9-21，updateLeadershipStatus 方法的执行步骤如下。

①如果 Master 节点之前不是领导（状态是 NOT_LEADER），当被选为领导（Leader）时，将其状态设置为 LEADER 并调用 masterInstance 的 electedLeader 方法将 Master 节点设置为 Leader。

②如果 Master 节点之前是领导（状态是 LEADER），当没有被选为 Leader 时，将其状态设置为 NOT_LEADER 并调用 masterInstance 的 revokedLeadership 方法撤销 Master 节点的领导关系。

8）isLeader：告知 ZooKeeperLeaderElectionAgent 所属的 Master 节点被选为 Leader，并更新领导关系状态。当 LeaderLatch 发现 Master 被选举为 Leader 后，会调用 ZooKeeperLeaderElectionAgent 的 isLeader 方法。isLeader 的实现如代码清单 9-22 所示。

代码清单9-22 被选为Leader

```
override def isLeader() {
  synchronized {
    if (!leaderLatch.hasLeadership) {
      return
    }

    logInfo("We have gained leadership")
    updateLeadershipStatus(true)
  }
}
```

9）notLeader：告知 ZooKeeperLeaderElectionAgent 所属的 Master 节点没有被选为 Leader，并更新领导关系状态。当 LeaderLatch 发现 Master 没有被选举为 Leader，会调用 ZooKeeperLeaderElectionAgent 的 notLeader 方法。notLeader 的实现如代码清单 9-23 所示。

代码清单9-23 没有被选为Leader

```
override def notLeader() {
```

```
    synchronized {
      if (leaderLatch.hasLeadership) {
        return
      }

      logInfo("We have lost leadership")
      updateLeadershipStatus(false)
    }
  }
```

小贴士：LeaderLatch 实际是通过在 Master 上创建领导选举的 Znode 节点，并对 Znode 节点添加监视点来发现 Master 是否成功“竞选”的。有关更多 ZooKeeper 节点、监视器的内容请访问 http://zookeeper.apache.org。

9.6 Master 详解

Master 是 local-cluster 部署模式和 Standalone 部署模式中，整个 Spark 集群最为重要的组件之一，它的设计将直接决定整个集群的可扩展性、可用性和容错性。Master 的职责包括 Worker 的管理、Application 的管理、Driver 的管理等。Master 负责对整个集群中所有资源的统一管理和分配，它接收各个 Worker 的注册、更新状态、心跳等消息，也接收 Driver 和 Application 的注册[⊖]。

Worker 向 Master 注册时会携带自身的身份和资源信息（如 ID、host、port、内核数、内存大小等），这些资源将按照一定的资源调度策略分配给 Driver 或 Application。Master 给 Driver 分配了资源后，将向 Worker 发送启动 Driver 的命令，后者在接收到启动 Driver 的命令后启动 Driver。Master 给 Application 分配了资源后，将向 Worker 发送启动 Executor 的命令，后者在接收到启动 Executor 的命令后启动 Executor。

Master 接收 Worker 的状态更新消息，用于“杀死”不匹配的 Driver 或 Application。

Worker 向 Master 发送的心跳消息有两个目的：一是告知 Master 自己还“活着”，另外则是某个 Master 出现故障后，通过领导选举选择了其他 Master 负责对整个集群的管理，此时被激活的 Master 可能并没有缓存 Worker 的相关信息，因此需要告知 Worker 重新向新的 Master 注册。

本节主要对 Master 进行详细分析，理解 local-cluster 部署模式和 Standalone 部署模式下 Master 如何对整个集群的资源进行管理和分配，但在此之前先需要按部就班地了解 Master 包含的属性。

- rpcEnv：即 RpcEnv。
- address：RpcEnv 的地址（即 RpcAddress）。RpcAddress 只包含 host 和 port 两个属

⊖ 由于注册 Driver 只发生在需要将 Driver 委托给 Spark 集群运行的情况下，对于 Driver 运行在客户端的情形，不会有注册 Driver 的行为。由于注册 Driver 与注册 Application 都是类似的，限于篇幅，本书只挑选注册 Application 进行介绍。

性，用来记录 Master URL 的 host 和 port。

- webUiPort：参数要指定的 WebUI 的端口。
- securityMgr：即 SecurityManager。
- conf：即 SparkConf。
- checkForWorkerTimeOutTask：检查 Worker 超时的任务。
- forwardMessageThread：包含一个线程的 ScheduledThreadPoolExecutor，启动的线程以 master-forward-message-thread 作为名称。forwardMessageThread 主要用于运行 checkForWorkerTimeOutTask 和 recoveryCompletionTask。
- hadoopConf：Hadoop 的配置。
- WORKER_TIMEOUT_MS：Worker 的超时时间。可通过 spark.worker.timeout 属性配置，默认为 60s。
- workers：所有注册到 Master 的 Worker 信息（WorkerInfo）的集合。
- REAPER_ITERATIONS：从 workers 中移除处于死亡（DEAD）状态的 Worker 所对应的 WorkerInfo 的权重。可通过 spark.dead.worker.persistence 属性配置，默认为 15。
- ECOVERY_MODE：恢复模式。可通过 spark.deploy.recoveryMode 属性配置，默认为 NONE。
- MAX_EXECUTOR_RETRIES：Executor 的最大重试数。可通过 spark.deploy.maxExecutorRetries 属性配置，默认为 10。
- idToApp：Application ID 与 ApplicationInfo 的映射关系。
- waitingApps：正等待调度的 Application 所对应的 ApplicationInfo 的集合。
- apps：所有 ApplicationInfo 的集合。
- idToWorker：Worker id 与 WorkerInfo 的映射关系。
- addressToWorker：Worker 的 RpcEnv 的地址（RpcAddress）与 WorkerInfo 的映射关系。
- endpointToApp：RpcEndpointRef 与 ApplicationInfo 的映射关系。
- addressToApp：Application 对应 Driver 的 RpcEnv 的地址（RpcAddress）与 ApplicationInfo 的映射关系。
- completedApps：已经完成的 ApplicationInfo 的集合。
- nextAppNumber：下一个 Application 的号码。nextAppNumber 将参与到 Application ID 的生成规则中。
- drivers：所有 Driver 信息（DriverInfo）的集合。
- completedDrivers：已经完成的 DriverInfo 的集合。
- RETAINED_APPLICATIONS：completedApps 中最多可以保留的 ApplicationInfo 的数量的限制大小。当 completedApps 中的 ApplicationInfo 数量大于等于 RETAINED_APPLICATIONS 时，需要对 completedApps 中的部分 ApplicationInfo 进行清

仅供非商业用途或交流学习使用

除。可通过 spark.deploy.retainedApplications 属性配置，默认为 200。

- RETAINED_DRIVERS：completedDrivers 中最多可以保留的 DriverInfo 的数量的限制大小。当 completedDrivers 中的 DriverInfo 数量大于等于 RETAINED_DRIVERS 时，需要对 completedDrivers 中的部分 DriverInfo 进行清除。可通过 spark.deploy.retainedDrivers 属性配置，默认为 200。
- waitingDrivers：正等待调度的 Driver 所对应的 DriverInfo 的集合。
- nextDriverNumber：下一个 Driver 的号码。
- masterMetricsSystem：实例名为 master 的 MetricsSystem，即 Master 的度量系统。masterMetricsSystem 是通过 MetricsSystem 的 createMetricsSystem 方法（已在 5.8 节详细介绍）创建的。
- applicationMetricsSystem：实例名为 applications 的 MetricsSystem，即应用程序的度量系统。applicationMetricsSystem 也是通过 MetricsSystem 的 createMetricsSystem 方法创建的。
- masterSource：类型为 MasterSource，是有关 Master 的度量来源。
- webUi：Master 的 WebUI，类型为 WebUI 的子类 MasterWebUI。
- masterPublicAddress：Master 的公开地址。可通过 Java 系统环境变量 SPARK_PUBLIC_DNS 配置，默认为 Master 的 RpcEnv 的地址。
- masterUrl：Master 的 Spark URL（即 spark://host:port 格式的地址）。
- masterWebUiUrl：Master 的 WebUI 的 URL。
- state：Master 所处的状态。Master 的状态包括支持（STANDBY）、激活（ALIVE）、恢复中（RECOVERING）、完成恢复（COMPLETING_RECOVERY）等。
- persistenceEngine：持久化引擎（PersistenceEngine）。
- leaderElectionAgent：领导选举代理（LeaderElectionAgent）。
- recoveryCompletionTask：当 Master 被选举为领导后，用于集群状态恢复的任务。
- spreadOutApps：是否允许 Application 能够在所有节点间调度。在所有节点间执行循环调度是 Spark 在实现更好的配置内存方法之前的临时解决方案，通过此方案可以避免 Application 总是固定在一小群节点上执行。可通过 spark.deploy.spreadOut 属性配置，默认为 true。
- defaultCores：应用程序默认的最大内核数。可通过 spark.deploy.defaultCores 属性配置，默认为 java.lang.Integer.MAX_VALUE。
- reverseProxy：SparkUI 是否采用反向代理。可通过 spark.ui.reverseProxy 属性配置，默认为 false。
- restServerEnabled：是否提供 REST 服务以提交应用程序。可通过 spark.master.rest.enabled 属性配置，默认为 true。
- restServer：REST 服务的实例，类型为 StandaloneRestServer。



仅供非商业用途或交流学习使用

- restServerBoundPort：REST 服务绑定的端口。

有了对 Master 属性的了解，现在可以来看看 Master 的主要执行流程了。

9.6.1 启动 Master

启动 Master 有作为 JVM 进程内的对象启动和作为单独的进程启动的两种方式。以对象启动的方式主要用于 local-cluster 模式，而作为进程启动则用于 Standalone 模式。

1. 对象方式启动

Master 的伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-24）用于创建 Master 对象，并将 Master 对象注册到 RpcEnv 中完成对 Master 对象的启动。

代码清单9-24 创建、启动Master

```
def startRpcEnvAndEndpoint(
    host: String,
    port: Int,
    webUiPort: Int,
    conf: SparkConf): (RpcEnv, Int, Option[Int]) = {
  val securityMgr = new SecurityManager(conf)
  val rpcEnv = RpcEnv.create(SYSTEM_NAME, host, port, conf, securityMgr)
  val masterEndpoint = rpcEnv.setupEndpoint(ENDPOINT_NAME,
    new Master(rpcEnv, rpcEnv.address, webUiPort, securityMgr, conf))
  val portsResponse = masterEndpoint.askWithRetry[BoundPortsResponse](Bound-
     PortsRequest)
  (rpcEnv, portsResponse.webUIPort, portsResponse.restPort)
}
```

根据代码清单 9-24，Master 的伴生对象的 startRpcEnvAndEndpoint 方法的执行步骤如下。

1）创建 SecurityManager。

2）创建 RpcEnv。5.3 节已对 RpcEnv 的 create 方法介绍过，此处不再赘述。

3）创建 Master，并且将 Master（Master 也继承了 ThreadSafeRpcEndpoint）注册到刚创建的 RpcEnv 中，并获得 Master 的 RpcEndpointRef。

4）通过 Master 的 RpcEndpointRef，向 Master 发送 BoundPortsRequest 消息，并获得返回的 BoundPortsResponse 消息。

5）返回创建的 RpcEnv、BoundPortsResponse 消息携带的 WebUIPort、REST 服务的端口（restPort）等信息。

在向 RpcEnv 中注册 Master 时，会触发对 Master 的 onStart 方法（见代码清单 9-25）的调用。

代码清单9-25 Master的onStart方法

```
override def onStart(): Unit = {
  logInfo("Starting Spark master at " + masterUrl)
```

仅供非商业用途或交流学习使用


```
    logInfo(s"Running Spark version ${org.apache.spark.SPARK_VERSION}")
    webUi = new MasterWebUI(this, webUiPort) // 创建Master的Web UI
    webUi.bind()
    masterWebUiUrl = "http://" + masterPublicAddress + ":" + webUi.boundPort
    if (reverseProxy) {
      masterWebUiUrl = conf.get("spark.ui.reverseProxyUrl", masterWebUiUrl)
      logInfo(s"Spark Master is acting as a reverse proxy. Master, Workers and " +
       s"Applications UIs are available at $masterWebUiUrl")
    }
    checkForWorkerTimeOutTask = forwardMessageThread.scheduleAtFixedRate(new
        Runnable {
      override def run(): Unit = Utils.tryLogNonFatalError {
        self.send(CheckForWorkerTimeOut) // 检查Worker超时
      }
    }, 0, WORKER_TIMEOUT_MS, TimeUnit.MILLISECONDS)

    if (restServerEnabled) { // 创建并启动REST服务
      val port = conf.getInt("spark.master.rest.port", 6066)
      restServer = Some(new StandaloneRestServer(address.host, port, conf, self,
          masterUrl))
    }
    restServerBoundPort = restServer.map(_.start())

    masterMetricsSystem.registerSource(masterSource)
    masterMetricsSystem.start() // 启动Master的度量系统
    applicationMetricsSystem.start() // 启动Application的度量系统
    masterMetricsSystem.getServletHandlers.foreach(webUi.attachHandler)
    applicationMetricsSystem.getServletHandlers.foreach(webUi.attachHandler)
    // 根据RECOVERY_MODE创建持久化引擎和领导选举代理
    val serializer = new JavaSerializer(conf)
    val (persistenceEngine_, leaderElectionAgent_) = RECOVERY_MODE match {
      case "ZOOKEEPER" =>
        logInfo("Persisting recovery state to ZooKeeper")
        val zkFactory =
          new ZooKeeperRecoveryModeFactory(conf, serializer)
        (zkFactory.createPersistenceEngine(), zkFactory.createLeaderElectionAgent(
            this))
      case "FILESYSTEM" =>
        val fsFactory =
          new FileSystemRecoveryModeFactory(conf, serializer)
        (fsFactory.createPersistenceEngine(), fsFactory.createLeaderElectionAgent(
            this))
      case "CUSTOM" =>
        val clazz = Utils.classForName(conf.get("spark.deploy.recoveryMode.factory"))
        val factory = clazz.getConstructor(classOf[SparkConf], classOf[Serializer])
          .newInstance(conf, serializer)
          .asInstanceOf[StandaloneRecoveryModeFactory]
        (factory.createPersistenceEngine(), factory.createLeaderElectionAgent(this))
      case _ =>
        (new BlackHolePersistenceEngine(), new MonarchyLeaderAgent(this))
    }
    persistenceEngine = persistenceEngine_
    leaderElectionAgent = leaderElectionAgent_
  }
```

仅供非商业用途或交流学习使用

根据代码清单 9-25，Master 的 onStart 方法的执行步骤如下。

1）创建 MasterWebUI，并绑定端口。由于 MasterWebUI 的原理与 4.3 节介绍的 SparkUI 类似，所以不再赘述。

2）拼接 MasterWebUI 的 URL，即 masterWebUiUrl。

3）如果启用了 Spark UI 的反向代理，那么将 masterWebUiUrl 设置为从 spark.ui.reverseProxyUrl 属性获得的反向代理的 URL。

4）启动对 Worker 超时进行检查的定时任务 checkForWorkerTimeOutTask，执行的时间间隔为 WORKER_TIMEOUT_MS。checkForWorkerTimeOutTask 实际上是向 Master 自身发送 CheckForWorkerTimeOut 消息完成检查的。

5）如果需要提供 REST 服务，那么创建并启动 StandaloneRestServer。StandaloneRestServer 的端口可通过 spark.master.rest.port 属性配置，默认为 6066。对 REST 服务感兴趣的读者可自行研究，本书不会对其过多介绍。

6）将 masterSource 注册到 masterMetricsSystem 后启动 masterMetricsSystem 和 app-lication-MetricsSystem，然后将 masterMetricsSystem 和 applicationMetricsSystem 的 Servlet-Context-Handler 添加到 MasterWebUI。

7）根据 RECOVERY_MODE 创建持久化引擎和领导选举代理。如果 RECOVERY_MODE 为 ZOOKEEPER，那么分别为 ZooKeeperPersistenceEngine 和 ZooKeeper-LeaderElectionAgent；如果 RECOVERY_MODE 为 FILESYSTEM，那么分别为 FileSystemPer-sistenceEngine 和 MonarchyLeaderAgent；如果 RECOVERY_MODE 为 CUSTOM，那么可通过 spark.deploy.recoveryMode.factory 属性配置创建持久化引擎和领导选举代理的工厂类，并创建对应的实例；如果 RECOVERY_MODE 为其他值时，则分别为 BlackHole-Persistence-Engine 和 MonarchyLeaderAgent。



2. 进程方式启动

Master 的伴生对象中实现了 main 方法，这样就可以作为单独的 JVM 进程启动了。main 方法的实现如代码清单 9-26 所示。

代码清单9-26 进程方式启动Master

```
def main(argStrings: Array[String]) {
  Utils.initDaemon(log)
  val conf = new SparkConf
  val args = new MasterArguments(argStrings, conf)
  val (rpcEnv, _, _) = startRpcEnvAndEndpoint(args.host, args.port, args.
    webUiPort, conf)
  rpcEnv.awaitTermination()
}
```

根据代码清单 9-26，main 方法的执行步骤如下。

1）创建 SparkConf。

仅供非商业用途或交流学习使用

2）创建 MasterArguments 以对执行 main 函数时传递的参数进行解析。在解析的过程中会将 Spark 属性配置文件中以 spark. 开头的属性保存到 SparkConf。

3）调用 Master 伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-24）创建并启动 Master 对象。

MasterArguments 用于解析系统环境变量和启动 Master 时指定的命令行参数，如代码清单 9-27 所示。

代码清单9-27 MasterArguments解析Master参数

```
private[master] class MasterArguments(args: Array[String], conf: SparkConf)
    extends Logging {
  var host = Utils.localHostName()
  var port = 7077
  var webUiPort = 8080
  var propertiesFile: String = null
  if (System.getenv("SPARK_MASTER_IP") != null) {
    logWarning("SPARK_MASTER_IP is deprecated, please use SPARK_MASTER_HOST")
    host = System.getenv("SPARK_MASTER_IP")
  }
  if (System.getenv("SPARK_MASTER_HOST") != null) {
    host = System.getenv("SPARK_MASTER_HOST")
  }
  if (System.getenv("SPARK_MASTER_PORT") != null) {
    port = System.getenv("SPARK_MASTER_PORT").toInt
  }
  if (System.getenv("SPARK_MASTER_WEBUI_PORT") != null) {
    webUiPort = System.getenv("SPARK_MASTER_WEBUI_PORT").toInt
  }
  parse(args.toList)
  propertiesFile = Utils.loadDefaultSparkProperties(conf, propertiesFile)
  if (conf.contains("spark.master.ui.port")) {
    webUiPort = conf.get("spark.master.ui.port").toInt
  }
```



根据代码清单 9-27，其中调用了 parse 方法来解析命令行参数，parse 的实现如下。

```
def parse(args: List[String]): Unit = args match {
  case ("--ip" | "-i") :: value :: tail =>
    Utils.checkHost(value, "ip no longer supported, please use hostname " + value)
    host = value
    parse(tail)
  case ("--host" | "-h") :: value :: tail =>
    Utils.checkHost(value, "Please use hostname " + value)
    host = value
    parse(tail)
  case ("--port" | "-p") :: IntParam(value) :: tail =>
    port = value
    parse(tail)
  case "--webui-port" :: IntParam(value) :: tail =>
    webUiPort = value
    parse(tail)
  case ("--properties-file") :: value :: tail =>
```

仅供非商业用途或交流学习使用


```
      propertiesFile = value
      parse(tail)
    case ("--help") :: tail =>
      printUsageAndExit(0)
    case Nil => {}
    case _ =>
      printUsageAndExit(1)
  }
```

根据上面的分析，MasterArguments 中解析的参数如表 9-1 所示。

表 9-1 MasterArguments 中解析的参数

属 性	含 义	默 认 值	系统环境变量	命令行参数
host	Master 的监听地址	机器名	SPARK_MASTER_IP（已弃用） SPARK_MASTER_HOST	--ip 或者 -i（已弃用） --host 或者 -h
port	Master 的监听端口	7077	SPARK_MASTER_PORT	--port 或者 -p
webUiPort	WebUI 的监听端口	8080	SPARK_MASTER_WEBUI_PORT	--webui-port
propertiesFile	Spark 属性文件	spark-defaults.conf	—	--properties-file

小贴士：命令行参数指定的值会覆盖系统环境变量指定的值。属性 spark.master.ui.port 指定的值会覆盖系统环境变量 SPARK_MASTER_WEBUI_PORT 或命令行参数 --webui-port 的值。



parse 函数解析的命令行参数如表 9-2 所示。

表 9-2 parse 函数解析的命令行参数

参 数 名	参 数 含 义
--ip 或者 -i	指定 host name，将来会停止使用，推荐使用 --host 或者 -h
--host 或者 -h	指定 host name
--port 或者 -p	指定端口
--webui-port	指定 WebUI 的端口
--properties-file	Spark 系统属性文件
--help	帮助

9.6.2 检查 Worker 超时

经过上一小节对启动 Master 的分析，我们知道定时任务 checkForWorkerTimeOutTask 是以 WORKER_TIMEOUT_MS 为时间间隔，通过不断向 Master 自身发送 CheckForWorkerTimeOut 消息来实现对 Worker 的超时检查的。Master 也继承自 RpcEndpoint，Master 实现的 receive 方法中处理 CheckForWorkerTimeOut 消息的代码如下。

仅供非商业用途或交流学习使用


```
case CheckForWorkerTimeOut =>
  timeOutDeadWorkers()
```

所以，Master 是通过调用 timeOutDeadWorkers 方法（见代码清单 9-28）处理 CheckForWorkerTimeOut 消息的。

代码清单9-28　检查Worker超时

```
private def timeOutDeadWorkers() {
  val currentTime = System.currentTimeMillis() // 下面过滤出所有超时的Worker
  val toRemove = workers.filter(_.lastHeartbeat < currentTime - WORKER_TIMEOUT_
     MS).toArray
  for (worker <- toRemove) {
    if (worker.state != WorkerState.DEAD) {
      logWarning("Removing %s because we got no heartbeat in %d seconds".format(
        worker.id, WORKER_TIMEOUT_MS / 1000))
      removeWorker(worker) // 移除Worker的相关信息
    } else {
      if (worker.lastHeartbeat < currentTime - ((REAPER_ITERATIONS + 1) * WORKER_
          TIMEOUT_MS)) {
        workers -= worker // 等待足够长的时间后将它从workers列表中移除
      }
    }
  }
}
```

根据代码清单 9-28，timeOutDeadWorkers 方法的处理步骤如下。

1）过滤出所有超时的 Worker，即当前时间与 WORKER_TIMEOUT_MS 之差仍然大于 WorkerInfo 的 lastHeartbeat 的 Worker 节点。

2）对所有超时的 Worker 进行如下处理。

① 如果 WorkerInfo 的状态不是 DEAD，则调用 removeWorker 方法（见代码清单 9-48）移除 Worker 的相关信息。

② 如果 WorkerInfo 的状态是 DEAD，则等待足够长的时间后将它从 workers 列表中移除。足够长的时间的计算公式为：REAPER_ITERATIONS 与 1 的和再乘以 WORKER_TIMEOUT_MS。

经过对检查 Worker 超时的分析，可以发现定时任务 checkForWorkerTimeOutTask 与 HeartbeatReceiver 的定时任务 timeoutCheckingTask 的原理十分类似，读者可以对比学习。

9.6.3　被选举为领导时的处理

Master 基于高可用性的考虑，可以同时启动多个 Master。这些 Master 中只有一个是激活（Active）状态的，其余的都是支持（Standby）状态。根据 9.5 节的介绍，Master 为了具备故障迁移的能力，它实现了 LeaderElectable 接口，因此当 Master 被选举为领导时，领导选举代理（LeaderElectionAgent）将会调用 Master 的 electedLeader 方法。electedLeader 方法的实现如代码清单 9-29 所示。

代码清单9-29 Master的electedLeader方法

```
override def electedLeader() {
  self.send(ElectedLeader)
}
```

根据代码清单 9-29，electedLeader 方法只是向 Master 自身发送了 ElectedLeader 消息。Master 的 receive 方法中实现了对 ElectedLeader 消息的处理，如代码清单 9-30 所示。

代码清单9-30 Master对ElectedLeader消息的处理

```
case ElectedLeader =>
  val (storedApps, storedDrivers, storedWorkers) = persistenceEngine.readPersisted-
     Data(rpcEnv)
  state = if (storedApps.isEmpty && storedDrivers.isEmpty && storedWorkers.
      isEmpty) {
    RecoveryState.ALIVE // 如果没有任何持久化信息，那么将Master的当前状态设置为激活
        (ALIVE)
  } else {
    RecoveryState.RECOVERING // 如果有持久化信息，那么将Master的当前状态设置为恢复中
        (RECOVERING)
  }
  logInfo("I have been elected leader! New state: " + state)
  if (state == RecoveryState.RECOVERING) {
  // 对整个集群的状态进行恢复
    beginRecovery(storedApps, storedDrivers, storedWorkers)
    recoveryCompletionTask = forwardMessageThread.schedule(new Runnable {
      override def run(): Unit = Utils.tryLogNonFatalError {
        self.send(CompleteRecovery)
      }
    }, WORKER_TIMEOUT_MS, TimeUnit.MILLISECONDS)
  }
```

根据代码清单 9-30，Master 处理 ElectedLeader 消息的步骤如下。

1）从持久化引擎中读取出持久化的 ApplicationInfo、DriverInfo、WorkerInfo 等信息。

2）如果没有任何持久化信息，那么将 Master 的当前状态设置为激活（ALIVE），否则将 Master 的当前状态设置为恢复中（RECOVERING）。

3）如果 Master 的当前状态为 RECOVERING，则调用 beginRecovery 方法（见代码清单 9-31）对整个集群的状态进行恢复。在集群状态恢复完成后，创建延时任务 recovery-CompletionTask，在常量 WORKER_TIMEOUT_MS 指定的时间后向 Master 自身发送 Complete-Recovery 消息。

代码清单9-31 恢复集群状态

```
private def beginRecovery(storedApps: Seq[ApplicationInfo], storedDrivers:
    Seq[DriverInfo],
    storedWorkers: Seq[WorkerInfo]) {
  for (app <- storedApps) { // 遍历从持久化引擎中读取的ApplicationInfo
    logInfo("Trying to recover app: " + app.id)
    try {
      registerApplication(app) // 注册ApplicationInfo
```

```
        app.state = ApplicationState.UNKNOWN
        app.driver.send(MasterChanged(self, masterWebUiUrl)) // 让Driver切换Master后重连
      } catch {
        case e: Exception => logInfo("App " + app.id + " had exception on
            reconnect")
      }
    }
    for (driver <- storedDrivers) { // 遍历从持久化引擎中读取的DriverInfo
      drivers += driver // 注册DriverInfo
    }
    for (worker <- storedWorkers) { // 遍历从持久化引擎中读取的WorkerInfo
      logInfo("Trying to recover worker: " + worker.id)
      try {
        registerWorker(worker) // 注册WorkerInfo
        worker.state = WorkerState.UNKNOWN
        // 让Worker切换Master后重连
        worker.endpoint.send(MasterChanged(self, masterWebUiUrl))
      } catch {
        case e: Exception => logInfo("Worker " + worker.id + " had exception on
            reconnect")
      }
    }
  }
```

根据代码清单 9-31，beginRecovery 方法的处理步骤如下。

1）遍历从持久化引擎中读取的 ApplicationInfo，对每个 ApplicationInfo 执行如下操作。

① 调用 registerApplication 方法（见代码清单 9-44）将 ApplicationInfo 添加到 apps、idToApp、endpointToApp、addressToApp、waitingApps 等缓存中。

② 将 ApplicationInfo 的状态设置为 UNKNOWN。

③ 向提交应用程序的 Driver 发送 MasterChanged 消息（此消息将携带被选举为领导的 Master 和此 Master 的 masterWebUiUrl 属性）。Driver 接收到 MasterChanged 消息后，将自身的 master 属性修改为当前 Master 的 RpcEndpointRef，并将 alreadyDisconnected 设置为 false，最后向 Master 发送 MasterChangeAcknowledged 消息（9.8.1 节将会对此介绍）。Master 接收到 MasterChangeAcknowledged 消息后将 ApplicationInfo 的状态修改为 WAITING，然后在不存在状态为 UNKNOWN 的 ApplicationInfo 和 WorkerInfo 时调用 completeRecovery 方法（见代码清单 9-32）完成恢复。

2）遍历从持久化引擎中读取的 DriverInfo，将每个 DriverInfo 添加到 drivers 缓存。

3）遍历从持久化引擎中读取的 WorkerInfo，对每个 WorkerInfo 执行如下操作。

① 调用 registerWorker 方法将 WorkerInfo 添加到 workers、idToWorker、addressToWorker 等缓存中。

② 将 WorkerInfo 的状态修改为 UNKNOWN。

③ 向 Worker 发送 MasterChanged 消息（此消息将携带被选举为领导的 Master 和此 Master 的 masterWebUiUrl 属性）。在 9.7.4 节中将介绍 Worker 接收到 MasterChanged 消息

后，将自身的activeMasterUrl、activeMasterWebUiUrl、master等属性修改为当前Master的对应信息，然后将connected设置为true，最后向Master发送WorkerSchedulerStateResponse消息。Master接收到WorkerSchedulerStateResponse消息后，首先将Worker-Info的状态修改为ALIVE，然后对此Worker上的Executor和Driver也进行恢复，最后在不存在状态为UNKNOWN的ApplicationInfo和WorkerInfo时调用completeRecovery方法（见代码清单9-32）完成恢复。

Master对ElectedLeader消息进行处理时创建的recoveryCompletionTask将向Master自身发送CompleteRecovery消息，Master的receive方法中对CompleteRecovery消息处理的代码如下。

```
case CompleteRecovery => completeRecovery()
```

从上述代码可以看到，Master接收到CompleteRecovery消息后只调用了completeRecovery方法。completeRecovery方法的实现如代码清单9-32所示。

代码清单9-32 完成集群恢复

```
private def completeRecovery() {
  if (state != RecoveryState.RECOVERING) { return }
  state = RecoveryState.COMPLETING_RECOVERY
  workers.filter(_.state == WorkerState.UNKNOWN).foreach(removeWorker)
  apps.filter(_.state == ApplicationState.UNKNOWN).foreach(finishApplication)
  drivers.filter(_.worker.isEmpty).foreach { d =>
    logWarning(s"Driver ${d.id} was not found after master recovery")
    if (d.desc.supervise) { // 由集群监管的Driver
      logWarning(s"Re-launching ${d.id}")
      relaunchDriver(d) // 重新调度运行指定的Driver
    } else {
      removeDriver(d.id, DriverState.ERROR, None)
      logWarning(s"Did not re-launch ${d.id} because it was not supervised")
    }
  }
  state = RecoveryState.ALIVE
  schedule()
  logInfo("Recovery complete - resuming operations!")
}
```

根据代码清单9-32，completeRecovery方法的执行步骤如下。

1）如果Master的状态不是RECOVERING，直接返回。

2）将workers中状态为UNKNOWN的所有WorkerInfo，通过调用removeWorker方法从Master中移除。

3）将apps中状态为UNKNOWN的所有ApplicationInfo，通过调用finishApplication方法从Master中移除。

4）从drivers中过滤出还没有分配Worker的所有DriverInfo，如果Driver是被监管的，则调用relaunchDriver方法重新调度运行指定的Driver，否则调用removeDriver方法移除Master维护的关于指定Driver的相关信息和状态。

5）将 Master 的状态设置为 ALIVE。

6）调用 schedule 方法进行资源调度。

小贴士：为什么状态为 UNKNOWN 的 WorkerInfo 或 ApplicationInfo 需要被移除？根据对 beginRecovery 方法的分析，Master 会向从持久化引擎中恢复的 Worker 或 Application 发送 MasterChanged 消息。Worker 或 Application 接收到 MasterChanged 消息后将向新的 Master 发送重连请求，Master 接收到重连请求后就知道 Worker 或 Application 还“活着”，于是更改它们的状态。对于已经停止或发生故障的 Worker 或 Application，Master 接收不到它们的重连请求，因此它们的状态依然是 UNKNOWN。

9.6.4 一级资源调度

Master 的资源调度是 Spark 的一级资源调度，分为对 Driver 的资源调度和对 Executor 的资源调度。schedule 方法是资源调度的入口方法，下面将通过它来深入分析 Master 的资源调度。

1. Driver 的资源调度

Master 的 schedule 方法主要完成对 Driver 的资源调度，其实现如代码清单 9-33 所示。

代码清单9-33 Driver的资源调度

```
private def schedule(): Unit = {
  if (state != RecoveryState.ALIVE) {
    return
  }
  val shuffledAliveWorkers = Random.shuffle(workers.toSeq.filter(_.state == Worker-
     State.ALIVE))
  val numWorkersAlive = shuffledAliveWorkers.size
  var curPos = 0
  for (driver <- waitingDrivers.toList) { // 遍历waitingDrivers中的DriverInfo
    var launched = false
    var numWorkersVisited = 0
    while (numWorkersVisited < numWorkersAlive && !launched) {
      val worker = shuffledAliveWorkers(curPos)
      numWorkersVisited += 1
      if (worker.memoryFree >= driver.desc.mem && worker.coresFree >= driver.
          desc.cores) {
        launchDriver(worker, driver) // 运行Driver
        waitingDrivers -= driver
        launched = true
      }
      curPos = (curPos + 1) % numWorkersAlive
    }
  }
  startExecutorsOnWorkers() // 在Worker上启动Executor
}
```

根据代码清单 9-33，schedule 方法的执行步骤如下。

1）如果 Master 的状态不是 ALIVE，那么直接返回。

2）过滤出 workers 中缓存的状态为 ALIVE 的 WorkerInfo，并随机洗牌，以避免 Driver 总是分配给小部分的 Worker。

3）遍历 waitingDrivers 中的 DriverInfo，对每个 DriverInfo 执行以下操作。

① 从第 2）步筛选的 WorkerInfo 中，挑出内存大小和内核数都满足 Driver 需要的。

② 调用 launchDriver 方法（见代码清单 9-34）运行 Driver。

③ 将 DriverInfo 从 waitingDrivers 中移除。

4）调用 startExecutorsOnWorkers 方法（见代码清单 9-35）在 Worker 上启动 Executor。

注意 Driver 一般运行在客户端，并不在 Spark 集群内。只有执行 Client.main 函数，将 org.apache.spark.deploy. ClientEndpoint 实例注册到 RpcEnv 中时，会构造 Driver 描述信息（DriverDescription）对象，并向 Master 发送 RequestSubmitDriver 消息，把 DriverDescription 对象传递给 Master，Master 将会根据 DriverDescription 的内容封装 DriverInfo 对象并注册到 waitingDrivers 中。

为使读者更容易理解 schedule 方法的实现，这里假设 Driver 所需的内存是 8（为简便起见，笔者选用简单的数字表示），CPU 内核数是 2，workers 中一共缓存了四个 WorkerInfo，那么对此 Driver 的资源调度如图 9-6 所示。

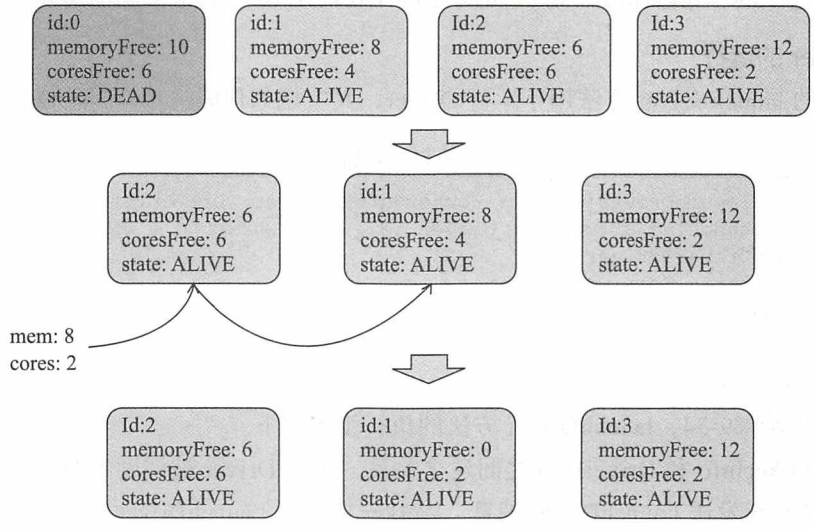


图 9-6 Driver 的资源调度过程示意图

这里对图 9-6 中的内容进行简单介绍：workers 中 id 为 0 的 WorkerInfo 的状态是 DEAD，因此将被过滤。在随机洗牌之后，假设 id 为 1 和 id 为 2 的 WorkerInfo 顺序上发生了互换，id 为 3 的还在最后。由于 id 为 2 的 WorkerInfo 的空闲内存不满足 Driver 的需求，

因此迭代查找下一个 WorkerInfo。id 为 1 的 WorkerInfo 的空闲内存和空闲 CPU 内核都满足 Driver 的需求，因此将在 id 为 1 的 Worker 上运行 Driver。由于 id 为 1 的 WorkerInfo 的内存和内核数分配给了 Driver，因此 id 为 1 的 WorkerInfo 剩余的空闲内存是 0，剩余的内核数是 2。

假设这时需要给另一个 Driver 进行资源调度，这个 Driver 所需的内存是 8，CPU 内核数是 4，那么资源调度过程可能如图 9-7 所示。

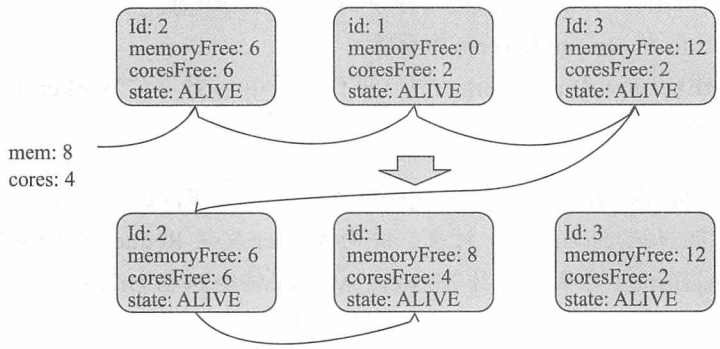


图 9-7　Driver 的资源调度过程示意图

图 9-7 中在第一次遍历过程中，发现所有可用的 WorkerInfo 都不满足 Driver 的资源需求，那么将进入下一次遍历。在第二次遍历之前，假设图 9-6 中的 Driver 已经释放了占用的资源给 id 为 1 的 WorkerInfo，那么 id 为 1 的 WorkerInfo 将满足资源的需要。

2. Driver 的运行

Master 的 launchDriver 方法用于运行 Driver，其实现如代码清单 9-34 所示。

代码清单9-34　运行Driver

```
private def launchDriver(worker: WorkerInfo, driver: DriverInfo) {
  logInfo("Launching driver " + driver.id + " on worker " + worker.id)
  worker.addDriver(driver)
  driver.worker = Some(worker)
  worker.endpoint.send(LaunchDriver(driver.id, driver.desc))
  driver.state = DriverState.RUNNING
}
```

根据代码清单 9-34，launchDriver 方法的执行步骤如下。

1）在 WorkerInfo 和 DriverInfo 之间建立关系，表示 Driver 被调度到 Worker 上运行。

2）向 Worker 发送 LaunchDriver 消息，Worker 接收到 LaunchDriver 消息后将运行 Driver（将在 9.7.5 节详细介绍）。

3）将 DriverInfo 的状态修改为 RUNNING。

3. Executor 的资源调度

schedule 方法的最后调用了 startExecutorsOnWorkers 方法，进而引发对 Application 的

Executor 资源的调度。startExecutorsOnWorkers 方法的实现如代码清单 9-35 所示。

代码清单9-35 Executor的资源调度

```
private def startExecutorsOnWorkers(): Unit = {
  for (app <- waitingApps if app.coresLeft > 0) {
    val coresPerExecutor: Option[Int] = app.desc.coresPerExecutor //获取每个
      Executor使用的内核数
    val usableWorkers = workers.toArray.filter(_.state == WorkerState.ALIVE)
      .filter(worker => worker.memoryFree >= app.desc.memoryPerExecutorMB &&
        worker.coresFree >= coresPerExecutor.getOrElse(1))
      .sortBy(_.coresFree).reverse
    val assignedCores = scheduleExecutorsOnWorkers(app, usableWorkers, spread-
      OutApps)
    for (pos <- 0 until usableWorkers.length if assignedCores(pos) > 0) {
      allocateWorkerResourceToExecutors(
        app, assignedCores(pos), coresPerExecutor, usableWorkers(pos))
    }
  }
}
```

根据代码清单 9-35，startExecutorsOnWorkers 方法遍历在 waitingApps 中还需要内核的 ApplicationInfo，并对其执行如下操作。

1）获取 Application 要求的每个 Executor 使用的内核数 coresPerExecutor。

2）找出 workers 中状态为 ALIVE、空闲空间满足 Application 要求的每个 Executor 使用的内存大小、空闲内核数满足 coresPerExecutor 的所有 WorkerInfo，并对这些 WorkerInfo 按照空闲内核数倒序排列，这样可以优先将应用分配给内核资源充足的 Worker。

3）调用 scheduleExecutorsOnWorkers 方法（见代码清单 9-36）在 Worker 上进行 Executor 的调度。scheduleExecutorsOnWorkers 方法返回在各个 Worker 上分配的内核数。

4）调用 allocateWorkerResourceToExecutors 方法（见代码清单 9-37）将 Worker 上的资源分配给 Executor。

代码清单9-36 scheduleExecutorsOnWorkers的实现

```
private def scheduleExecutorsOnWorkers(
    app: ApplicationInfo,
    usableWorkers: Array[WorkerInfo],
    spreadOutApps: Boolean): Array[Int] = {
  val coresPerExecutor = app.desc.coresPerExecutor
  val minCoresPerExecutor = coresPerExecutor.getOrElse(1)
  val oneExecutorPerWorker = coresPerExecutor.isEmpty
  val memoryPerExecutor = app.desc.memoryPerExecutorMB
  val numUsable = usableWorkers.length
  val assignedCores = new Array[Int](numUsable)
  val assignedExecutors = new Array[Int](numUsable)
  var coresToAssign = math.min(app.coresLeft, usableWorkers.map(_.coresFree).sum)

  def canLaunchExecutor(pos: Int): Boolean = {
    val keepScheduling = coresToAssign >= minCoresPerExecutor
    val enoughCores = usableWorkers(pos).coresFree - assignedCores(pos) >= min-
```

```
        CoresPerExecutor

      val launchingNewExecutor = !oneExecutorPerWorker || assignedExecutors(pos) == 0
      if (launchingNewExecutor) {
        val assignedMemory = assignedExecutors(pos) * memoryPerExecutor
        val enoughMemory = usableWorkers(pos).memoryFree - assignedMemory >= memory-
          PerExecutor
        val underLimit = assignedExecutors.sum + app.executors.size < app.executor-
          Limit
        keepScheduling && enoughCores && enoughMemory && underLimit
      } else {
        keepScheduling && enoughCores
      }
    }
    // 获取所有可以运行Executor的Worker的索引
    var freeWorkers = (0 until numUsable).filter(canLaunchExecutor)
    while (freeWorkers.nonEmpty) {
      freeWorkers.foreach { pos =>
        var keepScheduling = true
        while (keepScheduling && canLaunchExecutor(pos)) { // 分配内核、Executor
          coresToAssign -= minCoresPerExecutor
          assignedCores(pos) += minCoresPerExecutor

          if (oneExecutorPerWorker) {
            assignedExecutors(pos) = 1
          } else {
            assignedExecutors(pos) += 1
          }

          if (spreadOutApps) {
            keepScheduling = false
          }
        }
      }
      freeWorkers = freeWorkers.filter(canLaunchExecutor)
    }
    assignedCores
  }
```

在正式介绍 scheduleExecutorsOnWorkers 方法前，先对代码清单 9-36 中的一些变量进行说明。

- coresPerExecutor：Application 要求的每个 Executor 所需的内核数。
- minCoresPerExecutor：Application 要求的每个 Executor 所需的最小内核数。
- oneExecutorPerWorker：是否在每个 Worker 上只分配一个 Executor。当 Application 没有配置 coresPerExecutor 时，oneExecutorPerWorker 为 true。
- memoryPerExecutor：Application 要求的每个 Executor 所需的内存大小（单位为字节）。
- usableWorkers：缓存从 workers 中选出的状态为 ALIVE、空闲空间满足 ApplicationInfo 要求的每个 Executor 使用的内存大小、空闲内核数满足 coresPerExecutor 的所有 WorkerInfo。

- numUsable：可用的 Worker 的数量，即 usableWorkers 的大小。
- assignedCores：用于保存每个 Worker 给 Application 分配的内核数的数组。通过数组索引与 usableWorkers 中的 WorkerInfo 相对应。
- assignedExecutors：用于保存每个 Worker 给应用分配的 Executor 数的数组。通过数组索引与 usableWorkers 中的 WorkerInfo 相对应。
- coresToAssign：给 Application 要分配的内核数。coresToAssign 取 usableWorkers 中所有 WorkerInfo 的空闲内核数之和与 Application 还需的内核数中的最小值。

在 scheduleExecutorsOnWorkers 中还定义了一个名为 canLaunchExecutor 方法，用于判断 usableWorkers 中索引为参数 pos 指定的位置上的 WorkerInfo 是否能运行 Executor。判断的依据如下。

- coresToAssign >= minCoresPerExecutor。
- usableWorkers(pos).coresFree-assignedCores(pos)>= minCoresPerExecutor。

特别的，当 oneExecutorPerWorker 为 false 或 usableWorkers 中索引为参数 pos 指定的位置上的 WorkerInfo 还没有给 Application 分配任何 Executor，那么还需要以下依据。

- usableWorkers(pos).memoryFree >= (assignedExecutors(pos) + 1)* memoryPerExecutor。
- assignedExecutors.sum + app.executors.size < app.executorLimit。

有了以上的铺垫，现在来一起分析 scheduleExecutorsOnWorkers 方法的执行步骤。

1）获取所有可以运行 Executor 的 Worker 的索引 freeWorkers。

2）当 freeWorkers 不为空，那么遍历 freeWorkers 中的每一个索引位置，并进行如下操作。

① 由于给 Application 分配 Executor 时，每个 Executor 都至少需要 minCoresPerExecutor 指定大小的内核，因此将 coresToAssign 减去 minCoresPerExecutor 的大小。

② 将 usableWorkers 的 pos 位置的 WorkerInfo 已经分配的内核数（即 assignedCores 的 pos 位置的数字）增加 minCoresPerExecutor 的大小。

③ 如果 oneExecutorPerWorker 为 true，则将 assignedExecutors 的索引为 pos 的值设置为 1，否则将 assignedExecutors 的索引为 pos 的值增 1。

④ 如果 spreadOutApps 为 true，则将 keepScheduling 设置为 false，这会导致对 pos 位置上的 WorkerInfo 的资源调度提前结束，那么应用需要的其他 Executor 资源将会在其他 WorkerInfo 上调度。如果 spreadOutApps 为 false，那么应用需要的 Executor 资源将会不断从 pos 位置的 WorkerInfo 上调度，直到 pos 位置的 WorkerInfo 上的的资源被使用完。

3）返回 assignedCores。

为使读者更容易理解 startExecutorsOnWorkers 方法的实现，假设满足过滤条件并且已经按照内核数排序的 Worker 为 Worker0、Worker1、Worker2，Application 还需要的内核数（coresLeft）等于 7。Worker0 的内核数为 4，Worker1 的内核数为 4，Worker2 的内核数为 2。那么此时需要分配的内核数 coresToAssign 等于 math.min(7, (4 + 4 + 2))，即 coresToAssign 为 7。在 spreadOutApps 为 true 的情况下，此时的内核分配可以用图 9-8 表示。

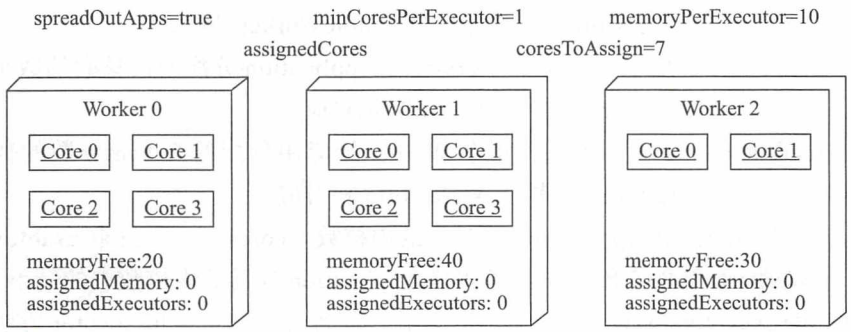


图 9-8 给应用分配 Executor 资源前

由于图 9-8 中的各个 Worker 都有足够的资源运行新的 Executor，所以在进行第一轮资源分配后的情况如图 9-9 所示。

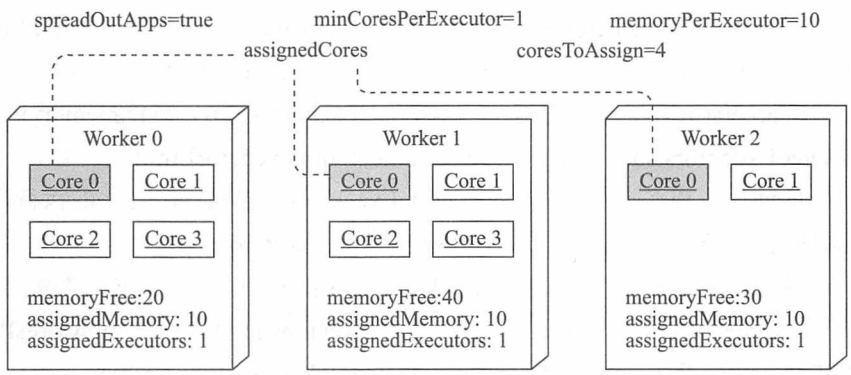


图 9-9 第一轮 Executor 资源分配后

由于图 9-9 中的各个 Worker 仍然都有足够的资源运行新的 Executor，所以在进行第二轮资源分配后的情况如图 9-10 所示。

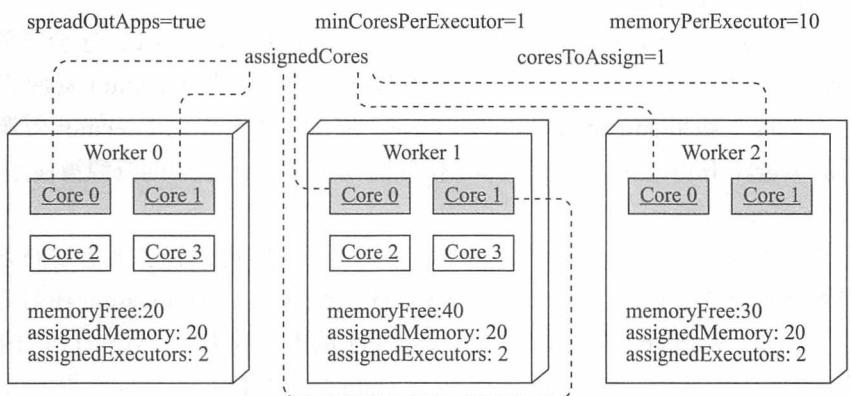


图 9-10 第二轮 Executor 资源分配后

此时，图 9-10 中的 Worker 0 没有了可用内存，Worker 2 则没有可用的内核，它们都没有充足的资源用来运行新的 Executor，所以在进行第三轮资源分配后的情况如图 9-11 所示。

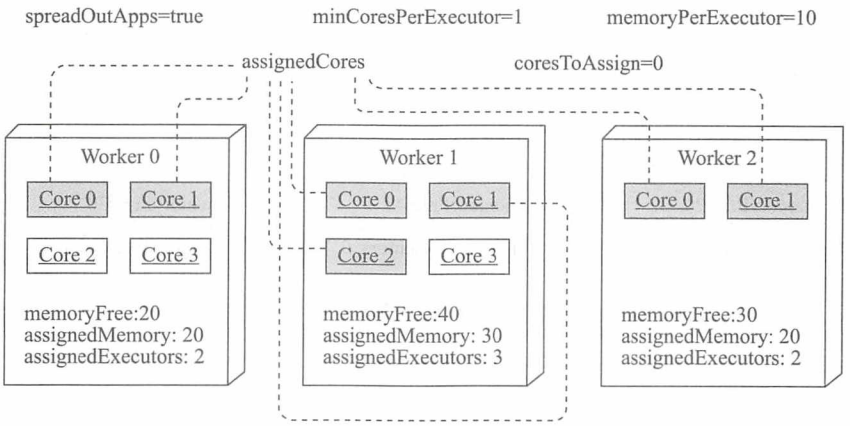


图 9-11 第三轮 Executor 资源分配后

图 9-11 显示，在第三轮 Executor 资源分配后，应用所需要的 7 个内核被分配完毕，不再需要从三个 Worker 上继续分配 Executor 了，应用的资源调度结束。

现在我们回到最开始的假设，在其他条件都不变的情况下，spreadOutApps 为 false，此时的内核分配可以用图 9-12 表示。

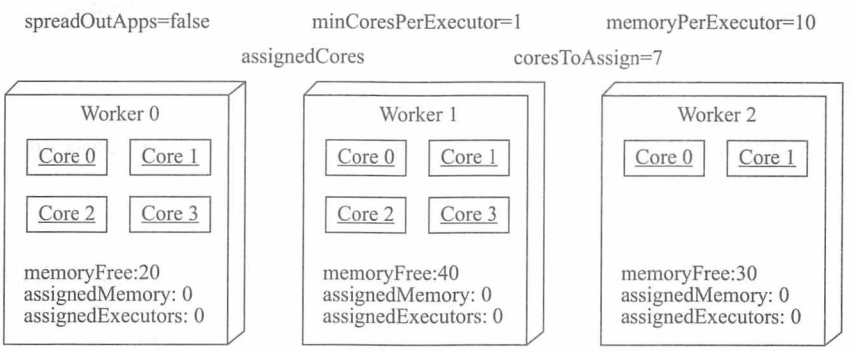


图 9-12 给应用分配 Executor 资源前

由于 spreadOutApps 为 false，所以将优先在 Worker 0 上分配 Executor 资源，分配之后如图 9-13 所示。

由于 spreadOutApps 为 false，将优先在 Worker 1 上分配 Executor 资源，分配之后如图 9-14 所示。

由于 spreadOutApps 为 false，将优先在 Worker 2 上分配 Executor 资源，分配之后如图 9-15 所示。

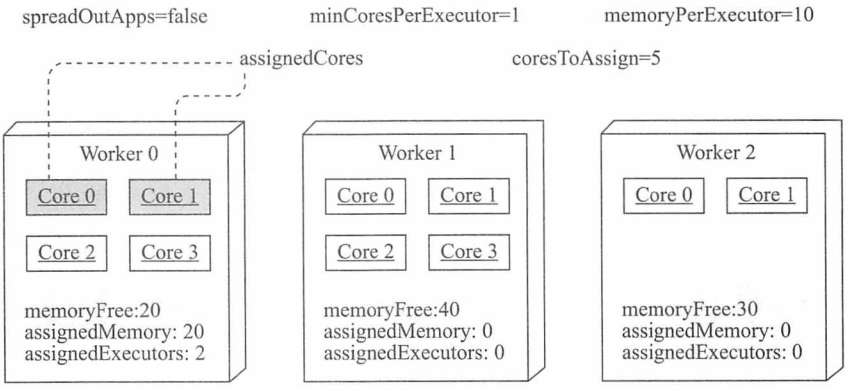


图 9-13　在 Worker 0 上分配 Executor 资源后

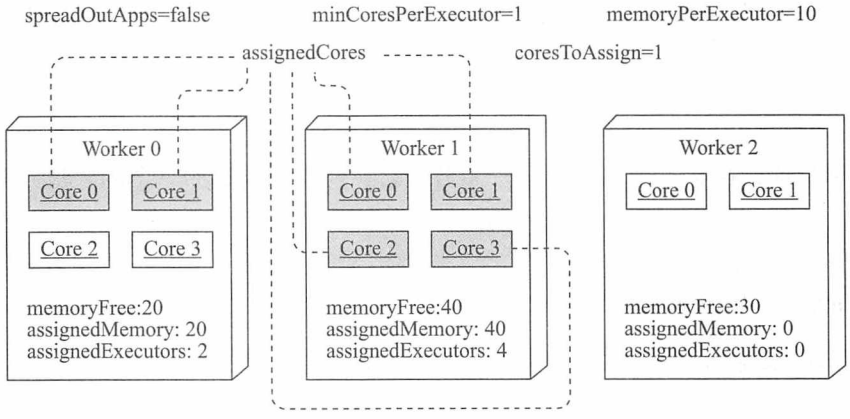


图 9-14　在 Worker 1 上分配 Executor 资源后

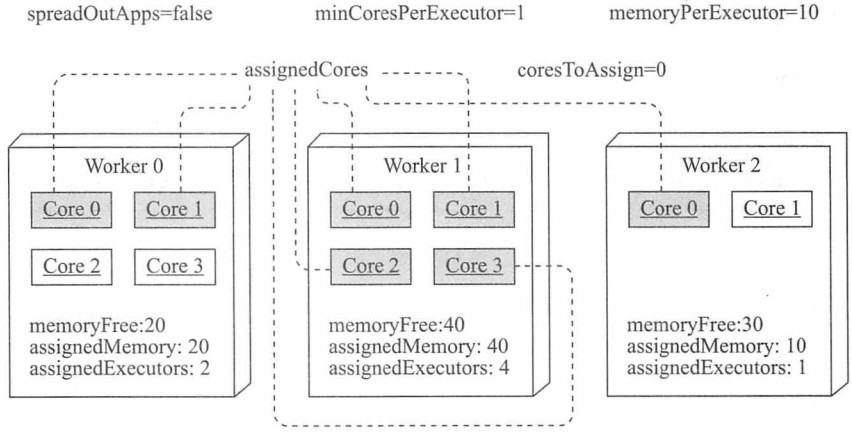


图 9-15　在 Worker 2 上分配 Executor 资源后

如图 9-15 所示，应用所需要的 7 个内核被分配完毕，不再需要从 3 个 Worker 上继续分配 Executor 了，应用的资源调度结束。

4. Executor 的运行

Master 的 allocateWorkerResourceToExecutors 方法用于将 Worker 的资源分配给 Executor，并运行 Executor，其实现如代码清单 9-37 所示。

代码清单9-37 将Worker的资源分配给Executor并运行

```
private def allocateWorkerResourceToExecutors(
    app: ApplicationInfo,
    assignedCores: Int,
    coresPerExecutor: Option[Int],
    worker: WorkerInfo): Unit = {
  val numExecutors = coresPerExecutor.map { assignedCores / _ }.getOrElse(1)
  val coresToAssign = coresPerExecutor.getOrElse(assignedCores)
  for (i <- 1 to numExecutors) {
    val exec = app.addExecutor(worker, coresToAssign)
    launchExecutor(worker, exec)
    app.state = ApplicationState.RUNNING
  }
}
```

根据代码清单 9-37，allocateWorkerResourceToExecutors 方法的执行步骤如下。

1）根据 Worker 分配给 Application 的内核数（assignedCores）与每个 Executor 需要的内核数（coresPerExecutor），计算在 Worker 上要运行的 Executor 数量（numExecutors）。如果没有指定 coresPerExecutor，说明 assignedCores 指定的所有内核都由一个 Executor 使用。

2）计算给 Executor 分配的内核数（coresToAssign）。如果指定了 coresPerExecutor，那么 coresToAssign 等于 coresPerExecutor，否则 coresToAssign 等于 assignedCores（即将所有内核分配给一个 Executor 使用）。

3）按照 numExecutors 的值，多次调用 launchExecutor 方法（见代码清单 9-38），在 Worker 上创建、运行 Executor，并将 ApplicationInfo 的状态设置为 RUNNING。

代码清单9-38 launchExecutor的实现

```
private def launchExecutor(worker: WorkerInfo, exec: ExecutorDesc): Unit = {
  logInfo("Launching executor " + exec.fullId + " on worker " + worker.id)
  worker.addExecutor(exec)
  worker.endpoint.send(LaunchExecutor(masterUrl,
    exec.application.id, exec.id, exec.application.desc, exec.cores, exec.memory))
  exec.application.driver.send(
    ExecutorAdded(exec.id, worker.id, worker.hostPort, exec.cores, exec.memory))
}
```

根据代码清单 9-38，launchExecutor 方法的执行步骤如下。

1）调用 WorkerInfo 的 addExecutor 方法添加新的 Executor 的信息。

2）向 Worker 发送 LaunchExecutor 消息（此消息携带着 masterUrl、Application 的 ID、Executor 的 ID、Application 的描述信息 ApplicationDescription、Executor 分配获得的内核数、Executor 分配获得的内存大小等信息）。Worker 对 LaunchExecutor 消息的处理将在

9.7.6 节详细分析。

3）向提交应用的 Driver 发送 ExecutorAdded 消息（此消息携带着 Executor 的 ID、Worker 的 ID、Worker 的 host 和 port、Executor 分配获得的内核数、Executor 分配获得的内存大小等信息）。ClientEndpoint 接收到 ExecutorAdded 消息后，将调用 StandaloneAppClientListener 的 executorAdded 方法。由于 StandaloneAppClientListener 只有 StandaloneSchedulerBackend 这一个实现类，因此实际上调用了 StandaloneSchedulerBackend 的 executorAdded 方法。StandaloneSchedulerBackend 的 executorAdded 方法只是打印 ExecutorAdded 消息携带的内容，所以不再展示其实现。

9.6.5 注册 Worker

在 Spark 集群中，Master 接收到 Driver 提交的应用程序后，需要根据应用的资源需求，将应用分配到 Worker 上去运行。一个集群刚开始的时候只有 Master，为了让后续启动的 Worker 加入到 Master 的集群中，每个 Worker 都需要在启动的时候向 Master 注册，Master 接收到 Worker 的注册信息后，将把 Worker 的各种重要信息（如 ID、host、port、内核数、内存大小等信息）缓存起来，以便进行资源的分配与调度。Master 为了容灾，还将 Worker 的信息通过持久化引擎进行持久化，以便经过领导选举出的新 Master 能够将集群的状态从错误或灾难中恢复。

Master 的 receiveAndReply 方法中实现了对 Worker 发送的 RegisterWorker 消息进行处理的实现，如代码清单 9-39 所示。

代码清单9-39　Master对RegisterWorker消息的处理

```
case RegisterWorker(
    id, workerHost, workerPort, workerRef, cores, memory, workerWebUiUrl) =>
  logInfo("Registering worker %s:%d with %d cores, %s RAM".format(
    workerHost, workerPort, cores, Utils.megabytesToString(memory)))
  if (state == RecoveryState.STANDBY) {
    context.reply(MasterInStandby)
  } else if (idToWorker.contains(id)) { // Worker重复注册
    context.reply(RegisterWorkerFailed("Duplicate worker ID"))
  } else {
    val worker = new WorkerInfo(id, workerHost, workerPort, cores, memory,
      workerRef, workerWebUiUrl) // 创建WorkerInfo
    if (registerWorker(worker)) { // 注册WorkerInfo
      persistenceEngine.addWorker(worker) // 持久化引擎对WorkerInfo持久化
      context.reply(RegisteredWorker(self, masterWebUiUrl))
      schedule()
    } else {
      val workerAddress = worker.endpoint.address
      logWarning("Worker registration failed. Attempted to re-register worker
          at same " +
        "address: " + workerAddress)
      context.reply(RegisterWorkerFailed("Attempted to re-register worker at
          same address: "
        + workerAddress))
```

```
        }
    }
```

根据代码清单 9-39，Master 处理 RegisterWorker 消息的步骤如下。

1）如果 Master 的状态是 STANDBY，那么向 Worker 回复 MasterInStandby 消息。

2）如果 idToWorker 中已经有了要注册的 Worker 的信息，那么说明 Worker 重复注册，Master 向 Worker 回复 RegisterWorkerFailed 消息。

3）如果 Master 的状态不是 STANDBY，且 idToWorker 中不包含要注册的 Worker 的信息，则执行如下操作。

① 利用 RegisterWorker 消息携带的信息创建 WorkerInfo。

② 调用 registerWorker 方法（见代码清单 9-40）注册 WorkerInfo。如果注册成功，那么对 WorkerInfo 进行持久化并向 Worker 回复 RegisteredWorker 消息，最后调用 schedule 方法（见代码清单 9-33）进行资源的调度。如果注册失败，则向 Worker 回复 RegisterWorkerFailed 消息。

registerWorker 方法用于将 WorkerInfo 注册到 Master 中，其实现如代码清单 9-40 所示。

代码清单9-40 注册WorkerInfo

```
private def registerWorker(worker: WorkerInfo): Boolean = {
  workers.filter { w =>
    (w.host == worker.host && w.port == worker.port) && (w.state == WorkerState.DEAD)
  }.foreach { w =>
    workers -= w
  }
  val workerAddress = worker.endpoint.address
  if (addressToWorker.contains(workerAddress)) {
    val oldWorker = addressToWorker(workerAddress)
    if (oldWorker.state == WorkerState.UNKNOWN) {
      removeWorker(oldWorker)
    } else {
      logInfo("Attempted to re-register worker at same address: " + workerAddress)
      return false
    }
  }

  workers += worker
  idToWorker(worker.id) = worker
  addressToWorker(workerAddress) = worker
  if (reverseProxy) {
     webUi.addProxyTargets(worker.id, worker.webUiAddress)
  }
  true
}
```

根据代码清单 9-40，registerWorker 方法的执行步骤如下。

1）从 workers 中移除 host 和 port 与要注册的 WorkerInfo 的 host 和 port 一样，且状态为 DEAD 的 WorkerInfo。

2）如果 addressToWorker 中包含地址相同的 WorkerInfo，并且此 WorkerInfo 的状态为 UNKNOWN，那么调用 removeWorker 方法（见代码清单 9-48），移除此 WorkerInfo 的相关状态，否则返回 false。

3）将要注册的 WorkerInfo 添加到 workers、idToWorker、addressToWorker 等缓存中。

4）对 WebUI 的反向代理进行设置。

9.6.6 更新 Worker 的最新状态

Worker 在向 Master 注册成功后，会向 Master 发送 WorkerLatestState 消息。WorkerLatestState 消息将携带 Worker 的身份标识、Worker 节点的所有 Executor 的描述信息、调度到当前 Worker 的所有 Driver 的身份标识。Master 接收到 WorkerLatestState 消息的处理如下。

```
case WorkerLatestState(workerId, executors, driverIds) =>
  idToWorker.get(workerId) match {
    case Some(worker) =>
      for (exec <- executors) {
        val executorMatches = worker.executors.exists {
          case (_, e) => e.application.id == exec.appId && e.id == exec.execId
        }
        if (!executorMatches) { //  “杀死”不匹配的Executor
          worker.endpoint.send(KillExecutor(masterUrl, exec.appId, exec.execId))
        }
      }

      for (driverId <- driverIds) {
        val driverMatches = worker.drivers.exists { case (id, _) => id == driverId }
        if (!driverMatches) {
          worker.endpoint.send(KillDriver(driverId)) //  “杀死”不匹配的Driver
        }
      }
    case None =>
      logWarning("Worker state from unknown worker: " + workerId)
  }
```

根据上述代码，Master 处理 WorkerLatestState 消息的步骤如下。

1）根据 Worker 的身份标识，从 idToWorker 取出注册的 WorkerInfo。

2）遍历 executors 中的每个 ExecutorDescription，与 WorkerInfo 的 executors 中保存的 ExecutorDesc 按照应用 ID 和 Executor ID 进行匹配。对于匹配不成功的，向 Worker 发送 KillExecutor 消息以“杀死”Executor。

3）遍历 driverIds 中的每个 Driver ID，与 WorkerInfo 的 drivers 中保存的 Driver ID 进行匹配。对于匹配不成功的，向 Worker 发送 KillDriver 消息以“杀死”Driver。

9.6.7 处理 Worker 的心跳

向 Master 注册 Worker，可以让 Master 知道 Worker 的资源配置，进而通过资源调度使得

仅供非商业用途或交流学习使用

Driver 及 Executor 可以在 Worker 上执行。如果 Worker 的 JVM 进程发生了崩溃或者 Worker 所在的机器宕机或网络不通，那么 Master 所维护的关于 Worker 的注册信息将变得不可用。为了让 Master 及时得知 Worker 的最新状态，Worker 需要向 Master 发送心跳，Master 将根据 Worker 的心跳更新 Worker 的最后心跳时间，以便为整个集群的健康工作提供参考。Master 的 receive 方法中实现了对 Worker 的心跳的处理，如代码清单 9-41 所示。

代码清单9-41 处理Worker的心跳

```
case Heartbeat(workerId, worker) =>
  idToWorker.get(workerId) match {
    case Some(workerInfo) =>
      workerInfo.lastHeartbeat = System.currentTimeMillis()
    case None =>
      if (workers.map(_.id).contains(workerId)) {
        logWarning(s"Got heartbeat from unregistered worker $workerId." +
          " Asking it to re-register.")
        worker.send(ReconnectWorker(masterUrl))
      } else {
        logWarning(s"Got heartbeat from unregistered worker $workerId." +
          " This worker was never registered, so ignoring the heartbeat.")
      }
  }
```

根据代码清单 9-41，Master 接收到 Heartbeat 消息后，将从 idToWorker 中找出缓存的 WorkerInfo，并将 WorkerInfo 的最后心跳时间（lastHeartbeat）更新为系统当前时间的时间戳。如果 idToWorker 中没有缓存的 WorkerInfo，且 workers 中有对应的 WorkerInfo（这说明定时任务 checkForWorkerTimeOutTask 检查到 Worker 超时，但是 WorkerInfo 的状态不是 DEAD，那么在调用 removeWorker 方法时将 WorkerInfo 从 idToWorker 中清除，此时的 workers 中仍然持有 WorkerInfo），那么向 Worker 发送 ReconnectWorker 消息。如果 idToWorker 中没有缓存的 WorkerInfo，且 workers 中也没有对应的 WorkerInfo，那么说明 checkForWorkerTimeOutTask 已经发现 Worker 很长时间没有心跳，并且 WorkerInfo 的状态为 DEAD 后，将 WorkerInfo 从 workers 中也移除了。



9.6.8 注册 Application

Driver 在启动后需要向 Master 注册 Application 信息，这样 Master 就能将 Worker 上的资源及 Executor 分配给 Driver。Driver 通过向 Master 发送 RegisterApplication 消息来注册 Application 信息，Master 因而也实现了对 RegisterApplication 消息的处理，如代码清单 9-42 所示。

代码清单9-42 注册Application

```
case RegisterApplication(description, driver) =>
  if (state == RecoveryState.STANDBY) {
    // ignore, don't send response
  } else {
```

仅供非商业用途或交流学习使用


```
    logInfo("Registering app " + description.name)
    val app = createApplication(description, driver) // 创建ApplicationInfo
    registerApplication(app) // 注册ApplicationInfo
    logInfo("Registered app " + description.name + " with ID " + app.id)
    persistenceEngine.addApplication(app) // 对ApplicationInfo持久化
    driver.send(RegisteredApplication(app.id, self))
    schedule()
  }
```

根据代码清单 9-42，Master 处理 RegisterApplication 消息的步骤如下。

1）如果 Master 当前的状态是 STANDBY，那么不作任何处理。

2）如果 Master 的状态不是 STANDBY，则执行如下操作。

① 利用 RegisterApplication 消息携带的信息调用 createApplication 方法（见代码清单 9-43）创建 ApplicationInfo。createApplication 方法中会给 Application 分配 ID。此 ID 的生成规则为 app-${yyyyMMddHHmmss}-${nextAppNumber}。

② 调用 registerApplication 方法（见代码清单 9-44）注册 ApplicationInfo。

③ 调用 PersistenceEngine 的 addApplication 方法对 ApplicationInfo 持久化。

④ 向 ClientEndpoint 发送 RegisteredApplication 消息（此消息将携带为 Application 生成的 ID 和 Master 自身的 RpcEndpointRef），以表示注册 Application 信息成功。

⑤ 调用 schedule 方法进行资源调度。



代码清单9-43 创建ApplicationInfo

```
private def createApplication(desc: ApplicationDescription, driver: RpcEndpointRef):
    ApplicationInfo = {
  val now = System.currentTimeMillis()
  val date = new Date(now)
  val appId = newApplicationId(date)
  new ApplicationInfo(now, appId, desc, date, driver, defaultCores)
}
```

代码清单9-44 注册ApplicationInfo

```
private def registerApplication(app: ApplicationInfo): Unit = {
  val appAddress = app.driver.address
  if (addressToApp.contains(appAddress)) {
    logInfo("Attempted to re-register application at same address: " + appAddress)
    return
  }

  applicationMetricsSystem.registerSource(app.appSource)
  apps += app
  idToApp(app.id) = app
  endpointToApp(app.driver) = app
  addressToApp(appAddress) = app
  waitingApps += app
  if (reverseProxy) {
    webUi.addProxyTargets(app.id, app.desc.appUiUrl)
  }
}
```

仅供非商业用途或交流学习使用

9.6.9 处理 Executor 的申请

如果 Driver 启用了 ExecutorAllocationManager，那么 ExecutorAllocationManager 将通过 StandaloneAppClient（将在 9.8 节介绍）中的 ClientEndpoint（将在 9.8.1 节介绍）向 Master 发送 RequestExecutors 消息请求 Executor。Master 接收并处理 RequestExecutors 消息的代码如下。

```
  case RequestExecutors(appId, requestedTotal) =>
context.reply(handleRequestExecutors(appId, requestedTotal))
```

Master 通过 handleRequestExecutors 方法处理 RequestExecutors 消息，其实现如下。

```
private def handleRequestExecutors(appId: String, requestedTotal: Int): Boolean = {
  idToApp.get(appId) match {
    case Some(appInfo) =>
      logInfo(s"Application $appId requested to set total executors to $requested-
          Total.")
      appInfo.executorLimit = requestedTotal
      schedule()
      true
    case None =>
      logWarning(s"Unknown application $appId requested $requestedTotal total
          executors.")
      false
  }
}
```



handleRequestExecutors 方法首先更改 ApplicationInfo 的 Executor 总数，然后调用 schedule 方法进行资源调度。

9.6.10 处理 Executor 的状态变化

Executor 在运行的整个生命周期中，会向 Master 不断发送 ExecutorStateChanged 消息报告自身的状态改变。Master 中实现了对 ExecutorStateChanged 消息的处理，以根据 Executor 的状态变化进行调整，代码如下。

```
case ExecutorStateChanged(appId, execId, state, message, exitStatus) =>
  val execOption = idToApp.get(appId).flatMap(app => app.executors.get(execId))
  execOption match {
    case Some(exec) =>
      val appInfo = idToApp(appId)
      val oldState = exec.state
      exec.state = state

      if (state == ExecutorState.RUNNING) {
        assert(oldState == ExecutorState.LAUNCHING,
          s"executor $execId state transfer from $oldState to RUNNING is illegal")
        appInfo.resetRetryCount()
      }

      exec.application.driver.send(ExecutorUpdated(execId, state, message, exit-
          Status, false))
```

仅供非商业用途或交流学习使用


```
      if (ExecutorState.isFinished(state)) {
        logInfo(s"Removing executor ${exec.fullId} because it is $state")
        if (!appInfo.isFinished) {
          appInfo.removeExecutor(exec)
        }
        exec.worker.removeExecutor(exec)

        val normalExit = exitStatus == Some(0)
        if (!normalExit
            && appInfo.incrementRetryCount() >= MAX_EXECUTOR_RETRIES
            && MAX_EXECUTOR_RETRIES >= 0) { // < 0 disables this application-
                killing path
          val execs = appInfo.executors.values
          if (!execs.exists(_.state == ExecutorState.RUNNING)) {
            logError(s"Application ${appInfo.desc.name} with ID ${appInfo.id}
                failed " +
              s"${appInfo.retryCount} times; removing it")
            removeApplication(appInfo, ApplicationState.FAILED)
          }
        }
      }
      schedule()
    case None =>
      logWarning(s"Got status update for unknown executor $appId/$execId")
  }
```

根据上述代码，Master 收到 ExecutorStateChanged 消息后，将针对 Executor 的运行中（RUNNING）、完成（包括 KILLED、FAILED、LOST、EXITED 等状态）、正常退出、异常退出等状态进行相应的处理。如果 Executor 非正常退出且重试次数已经超过了 MAX_EXECUTOR_RETRIES 的限制，那么 Executor 所属的 Application 在没有任何 Executor 处于 RUNNING 状态时将被彻底移除。



9.6.11 Master 的常用方法

除了以上介绍的工作流程外，Master 还有一些值得注意的功能，包括回收 Master 的领导权、移除 Driver、重新运行 Driver、移除 Worker、移除 Application 等。下面将对这些功能的关键代码进行分析。

1. revokedLeadership

根据 9.5 节对 ZooKeeperLeaderElectionAgent 的介绍，我们知道当 Master 没有被选举为领导时，ZooKeeperLeaderElectionAgent 将会调用 revokedLeadership 方法撤销 Master 的领导关系，其实现如代码清单 9-45 所示。

代码清单9-45　撤销Master的领导关系

```
override def revokedLeadership() {
  self.send(RevokedLeadership)
}
```

仅供非商业用途或交流学习使用

根据代码清单 9-45，revokedLeadership 方法只是向 Master 自身发送了 RevokedLeadership 消息。

Master 实现的 receive 方法中实现了对 RevokedLeadership 消息的处理，代码如下。

```
case RevokedLeadership =>
  logError("Leadership has been revoked -- master shutting down.")
  System.exit(0)
```

根据上述代码，Master 将退出。

2. removeDriver

removeDriver 方法用于移除 Master 维护的关于指定 Driver 的相关信息和状态，其实现如代码清单 9-46 所示。

代码清单9-46 removeDriver方法的实现

```
private def removeDriver(
    driverId: String,
    finalState: DriverState,
    exception: Option[Exception]) {
  drivers.find(d => d.id == driverId) match { //从DriverInfo的集合drivers中找到指定
      的DriverInfo
    case Some(driver) =>
      logInfo(s"Removing driver: $driverId")
      drivers -= driver // 从drivers中移除找到的DriverInfo
      if (completedDrivers.size >= RETAINED_DRIVERS) {
        val toRemove = math.max(RETAINED_DRIVERS / 10, 1)
        completedDrivers.trimStart(toRemove)  //  移除completedDrivers中开头的一些
            DriverInfo
      }
      completedDrivers += driver // 将找到的DriverInfo放入completedDrivers集合中
      persistenceEngine.removeDriver(driver) // 移除DriverInfo的持久化数据
      driver.state = finalState
      driver.exception = exception
      driver.worker.foreach(w => w.removeDriver(driver))
      schedule() // 由于腾出了Driver占用的资源，所以对其他Application和Driver进行调度
    case None =>
      logWarning(s"Asked to remove unknown driver: $driverId")
  }
}
```

根据代码清单 9-46，removeDriver 方法的执行步骤如下。

1）从 DriverInfo 的集合 drivers 中找到指定的 DriverInfo。

2）从 drivers 中移除找到的 DriverInfo。

3）如果已经完成的 DriverInfo 的集合 completedDrivers 中的元素数量大于等于 RETAINED_DRIVERS，那么移除 completedDrivers 中开头的一些 DriverInfo。

4）将找到的 DriverInfo 放入 completedDrivers 集合中。

5）调用持久化引擎的 removeDriver 方法移除 DriverInfo 的持久化数据。

6）对 DriverInfo 的一些属性进行修改，最后对处理此 Driver 提交的 Job 的所有 Worker，

仅供非商业用途或交流学习使用

调用 WorkerInfo 的 removeDriver 方法（实现很简单，感兴趣的读者请自行阅读），移除 Worker 处理的 Driver 信息。

7）由于 Worker 上的资源已经腾出来了，所以需要调用 schedule 方法对其他 Application 和 Driver 进行调度。

3. relaunchDriver

relaunchDriver 方法用于重新调度运行指定的 Driver，其实现如代码清单 9-47 所示。

代码清单9-47 重新调度运行Driver

```
private def relaunchDriver(driver: DriverInfo) {
  driver.worker = None // 表示Driver提交的应用没有被任何Worker处理
  driver.state = DriverState.RELAUNCHING
  waitingDrivers += driver
  schedule()
}
```

根据代码清单 9-47，relaunchDriver 方法的执行步骤如下。

1）将 DriverInfo 的 worker 属性置为 None，以表示 Driver 提交的应用没有被任何 Worker 处理。

2）将 DriverInfo 的状态置为重新运行（RELAUNCHING）。

3）将 DriverInfo 放入所有等待调度的 Driver 的集合 waitingDrivers 中。

4）调用 schedule 方法对 Driver 重新调度运行。



4. removeWorker

removeWorker 方法用于移除 Master 维护的关于指定 Worker 的相关信息和状态，其实现如代码清单 9-48 所示。

代码清单9-48 removeWorker的实现

```
private def removeWorker(worker: WorkerInfo) {
  logInfo("Removing worker " + worker.id + " on " + worker.host + ":" + worker.port)
  worker.setState(WorkerState.DEAD)
  idToWorker -= worker.id
  addressToWorker -= worker.endpoint.address
  if (reverseProxy) {
    webUi.removeProxyTargets(worker.id)
  }
  for (exec <- worker.executors.values) { // 将Worker上的Executor都作为丢失状态处理
    logInfo("Telling app of lost executor: " + exec.id)
    exec.application.driver.send(ExecutorUpdated(
      exec.id, ExecutorState.LOST, Some("worker lost"), None, workerLost = true))
    exec.state = ExecutorState.LOST
    exec.application.removeExecutor(exec)
  }
  for (driver <- worker.drivers.values) {
    if (driver.desc.supervise) {
      logInfo(s"Re-launching ${driver.id}")
      relaunchDriver(driver) // 重新调度运行Driver
```

仅供非商业用途或交流学习使用


```
      } else {
        logInfo(s"Not re-launching ${driver.id} because it was not supervised")
        removeDriver(driver.id, DriverState.ERROR, None)
      }
    }
    persistenceEngine.removeWorker(worker)
  }
```

根据代码清单 9-48，removeWorker 方法的执行步骤如下。

1）将 WorkerInfo 的状态置为 DEAD。

2）从 idToWorker 和 addressToWorker 等缓存中移除 WorkerInfo。

3）向此 Worker 的所有 Executor 所服务的 Drvier Application 发送 ExecutorUpdated 消息，更新 Executor 的状态为 LOST。最后调用 ApplicationInfo 的 removeExecutor 方法（此方法的实现很简单，感兴趣的读者可以自行阅读），移除分配给 Application 的 Executor 的描述信息。

4）对 Worker 所处理的所有 Driver 进行如下操作：如果 Driver 是被监管的，则调用 relaunchDriver 方法（见代码清单 9-47）重新调度运行 Driver，否则调用 removeDriver 方法（见代码清单 9-46）移除 Driver 的相关信息和状态。

5）调用持久化引擎的 removeWorker 方法移除 WorkerInfo 的持久化数据。

5. removeApplication

removeApplication 方法用于移除 Master 中缓存的 Application 及 Application 相关的 Driver 信息，其实现如代码清单 9-49 所示。

代码清单9-49 removeApplication方法的实现

```
def removeApplication(app: ApplicationInfo, state: ApplicationState.Value) {
  if (apps.contains(app)) {
    logInfo("Removing app " + app.id)
    apps -= app
    idToApp -= app.id
    endpointToApp -= app.driver
    addressToApp -= app.driver.address
    if (reverseProxy) {
      webUi.removeProxyTargets(app.id)
    }
    if (completedApps.size >= RETAINED_APPLICATIONS) {
      val toRemove = math.max(RETAINED_APPLICATIONS / 10, 1)
      completedApps.take(toRemove).foreach { a =>
        applicationMetricsSystem.removeSource(a.appSource)
      }
      completedApps.trimStart(toRemove)
    }
    completedApps += app // 将ApplicationInfo作为已完成的历史应用
    waitingApps -= app

    for (exec <- app.executors.values) { //遍历分配给Application的Executor
      killExecutor(exec) //  “杀死” Executor
```

仅供非商业用途或交流学习使用


```
        }
        app.markFinished(state)
        if (state != ApplicationState.FINISHED) {
          app.driver.send(ApplicationRemoved(state.toString))
        }
        persistenceEngine.removeApplication(app) // 移除ApplicationInfo的持久化信息
        schedule()

        workers.foreach { w =>
          w.endpoint.send(ApplicationFinished(app.id))
        }
      }
    }
```

根据代码清单 9-49，removeApplication 方法的执行步骤如下。

1）移除 Master 中缓存的 ApplicationInfo 及 ApplicationInfo 相关的 DriverInfo。

2）将 ApplicationInfo 添加到数组缓冲 completedApps 中。

3）遍历分配给 Application 的 Executor，调用 killExecutor 方法“杀死”。

4）向 DriverEndpoint 发送 ApplicationRemoved 消息，告诉 Driver 驱动的 Application 已经被移除。

5）对 ApplicationInfo 去持久化。

6）调用 schedule 方法进行资源调度。

7）向每个 Worker 发送 ApplicationFinished 消息，以告知 Application 已完成。



9.7 Worker 详解

Worker 是 Spark 在 local-cluster 部署模式和 Standalone 部署模式中对工作节点的资源和 Executor 进行管理的服务。Worker 一方面向 Master 汇报自身所管理的资源信息，一方面接收 Master 的命令运行 Driver 或者为 Application 运行 Executor。同一个机器上可以同时部署多个 Worker 服务，一个 Worker 也可以启动多个 Executor。当 Executor 完成后，Worker 将回收 Executor 使用的资源。

本节将对 Worker 进行详细分析，但是我们还是从了解它所包含的属性开始。Worker 包含的属性如下。

- rpcEnv：即 RpcEnv。
- webUiPort：参数指定的 WebUI 的端口。
- cores：内核数。
- memory：内存大小。
- masterRpcAddresses：Master 的 RpcEnv 地址（即 RpcAddress）的数组。由于一个集群为了可靠性和容错，需要多个 Master 节点，因此用数组来存储它们的 RpcEnv 地址。
- endpointName：Worker 注册到 RpcEnv 的名称。

仅供非商业用途或交流学习使用

- workDirPath：字符串表示的 Worker 的工作目录。
- conf：即 SparkConf。
- securityMgr：即 SecurityManager。
- host：Worker 的 RpcEnv 的 host。
- port：Worker 的 RpcEnv 的端口。
- forwordMessageScheduler：用于发送消息的调度执行器（ScheduledThreadPoolExecutor）。forwordMessageScheduler 只能调度执行一个线程，执行的线程以 worker-forward-message-scheduler 作为名称。
- cleanupThreadExecutor：通过 Executors 的 newSingleThreadExecutor 方法创建的线程执行器，用于清理 Worker 的工作目录。由 cleanupThreadExecutor 执行的线程以 worker-cleanup-thread 作为名称。
- HEARTBEAT_MILLIS：向 Master 发送心跳的时间间隔，是 spark.worker.timeout 属性值的 1/4，默认为 15 秒。
- INITIAL_REGISTRATION_RETRIES：此常量固定为 6，代表连接 Master 的前六次尝试。
- TOTAL_REGISTRATION_RETRIES：此常量固定为 16，代表连接 Master 最多有 16 次尝试。
- FUZZ_MULTIPLIER_INTERVAL_LOWER_BOUND：此常量固定为 0.500，是为了确保尝试的时间间隔不至于过小的下边界。
- REGISTRATION_RETRY_FUZZ_MULTIPLIER：0～1 范围的随机数与 FUZZ_MULTIPLIER_INTERVAL_LOWER_BOUND 的和，所以 REGISTRATION_RETRY_FUZZ_MULTIPLIER 的真实范围在 0.5～1.5 之间。加入随机数是为了避免各个 Worker 在同一时间发送心跳。
- INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS：代表前六次尝试的时间间隔。INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS 取 10 和 REGISTRATION_RETRY_FUZZ_MULTIPLIER 乘积的最近整数，因此这六次尝试的时间间隔在 5～15s 之间。
- PROLONGED_REGISTRATION_RETRY_INTERVAL_SECONDS：代表最后十次尝试的时间间隔。PROLONGED_REGISTRATION_RETRY_INTERVAL_SECONDS 取 60 和 REGISTRATION_RETRY_FUZZ_MULTIPLIER 乘积的最近整数，因此最后十次尝试的时间间隔在 30～90s 之间。
- CLEANUP_ENABLED：是否对旧的 Application 产生的文件夹及文件进行清理。可通过 spark.worker.cleanup.enabled 属性配置，默认为 false。
- CLEANUP_INTERVAL_MILLIS：对旧的 Application 产生的文件夹及文件进行清理的时间间隔。可通过 spark.worker.cleanup.interval 属性配置，默认为 1800 秒。



- APP_DATA_RETENTION_SECONDS：Application 产生的文件夹及文件保留的时间。可通过 spark.worker.cleanup.appDataTtl 属性配置，默认为 7 天。
- master：Master 的 RpcEndpointRef。
- activeMasterUrl：处于激活（ALIVE）状态的 Master 的 URL。
- activeMasterWebUiUrl：处于激活（ALIVE）状态的 Master 的 WebUI 的 URL。
- workerWebUiUrl：Worker 的 WebUI 的 URL。
- workerUri：Worker 的 URL，格式为 spark://$name@${ host}:${ port}。
- registered：标记 Worker 是否已经注册到 Master。
- connected：标记 Worker 是否已经连接到 Master。
- workerId：Worker 的身份标识。
- sparkHome：环境变量 SPARK_HOME 的值。
- workDir：由 java.io.File 表示的 Worker 的工作目录。
- finishedExecutors：已经完成的 Executor 的身份标识与 ExecutorRunner 之间的映射关系。
- drivers：Driver 的身份标识与 DriverRunner 之间的映射关系。
- executors：Executor 的身份标识与 ExecutorRunner 之间的映射关系。
- finishedDrivers：已经完成的 Driver 的身份标识与 DriverRunner 之间的映射关系。
- appDirectories：Application 的 ID 与对应的目录集合之间的映射关系。
- finishedApps：已经完成的 Application 的 ID 的集合。
- retainedExecutors：保留的 Executor 的数量。可通过 spark.worker.ui.retainedExecutors 属性配置，默认为 1000。
- retainedDrivers：保留的 Driver 的数量。可通过 spark.worker.ui.retainedDrivers 属性配置，默认为 1000。
- shuffleService：外部的 Shuffle 服务。在 6.9 节曾经简单介绍过外部 Shuffle 服务的客户端 ExternalShuffleClient。这里则是外部 Shuffle 服务的服务端实现类 ExternalShuffleService。这里虽然创建了 ExternalShuffleService，但是只有配置了外部的 Shuffle 服务时，才会启动它。[⊖]
- publicAddress：Worker 的公共地址，如果设置了环境变量 SPARK_PUBLIC_DNS，则为环境变量 SPARK_PUBLIC_DNS 的值，否则为 host。
- webUi：Worker 的 WebUI，类型为 WebUI 的子类 WorkerWebUI。
- connectionAttemptCount：连接尝试次数的计数器。
- metricsSystem：实例名为 worker 的 MetricsSystem，即 Worker 的度量系统。
- workerSource：类型为 WorkerSource，是有关 Worker 的度量来源。

⊖ 本书不会对 ExternalShuffleClient 和 ExternalShuffleService 的实现进行深入分析，留给感兴趣的读者自行研究。

- registerMasterFutures：由于 Worker 向 Master 进行注册的过程是异步的，此变量保存线程返回的 java.util.concurrent.Future。
- registrationRetryTimer：Worker 向 Master 进行注册重试的定时器。
- registerMasterThreadPool：Worker 向 Master 进行注册的线程池。此线程池的大小与 Master 节点的数量一样，启动的线程以 worker-register-master-threadpool 为前缀。
- coresUsed：当前 Worker 已经使用的内核数。Worker 提供了 coresFree 方法返回 cores 属性和 coresUsed 的差值，作为空闲的内核数。
- memoryUsed：当前 Worker 已经使用的内存大小。Worker 提供了 memoryFree 方法返回 memory 属性和 memoryUsed 的差值，作为空闲的内存大小。

有了对 Worker 中包含属性的了解，现在可以来看看 Worker 的主要执行流程了。

9.7.1 启动 Worker

启动 Worker 有作为 JVM 进程内的对象启动和作为单独的进程启动两种方式。以对象启动的方式主要用于 local-cluster 模式，而作为进程启动则用于 Standalone 模式。

1. 对象方式启动

Worker 的伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-50）用于创建、启动 Worker。

代码清单9-50 创建、启动Worker

```
def startRpcEnvAndEndpoint(
    host: String, port: Int, webUiPort: Int, cores: Int, memory: Int,
    masterUrls: Array[String], workDir: String,
    workerNumber: Option[Int] = None, conf: SparkConf = new SparkConf): RpcEnv = {

  val systemName = SYSTEM_NAME + workerNumber.map(_.toString).getOrElse("")
  val securityMgr = new SecurityManager(conf)
  val rpcEnv = RpcEnv.create(systemName, host, port, conf, securityMgr)
  val masterAddresses = masterUrls.map(RpcAddress.fromSparkURL(_))
  rpcEnv.setupEndpoint(ENDPOINT_NAME, new Worker(rpcEnv, webUiPort, cores, memory,
    masterAddresses, ENDPOINT_NAME, workDir, conf, securityMgr))
  rpcEnv
}
```

根据代码清单 9-50，Worker 的伴生对象的 startRpcEnvAndEndpoint 方法的执行步骤如下。

1）生成 Worker 的 RpcEnv 的名称。生成规则为 SYSTEM_NAME 后跟 Worker 号。由于常量 SYSTEM_NAME 的值为 sparkWorker，所以 Worker 的 RpcEnv 的名称将会是 sparkWorker0、sparkWorker1 等。

2）创建 SecurityManager。

3）创建 Worker 的 RpcEnv。

4）将所有 Master 的 Spark URL（格式为 spark://host:port）转换为 RpcAddress 地址。Rpc-

Address 只包含 host 和 port 两个属性。

5）创建 Worker，并且将 Worker（Worker 也继承了 ThreadSafeRpcEndpoint）注册到刚创建的 RpcEnv 中。

6）返回 Worker 的 RpcEnv。

在向 RpcEnv 中注册 Worker 时，会触发对 Worker 的 onStart 方法（见代码清单 9-51）的调用。

代码清单9-51 Worker的onStart方法

```
override def onStart() {
  assert(!registered)
  // 省略日志输出代码
  createWorkDir()
  shuffleService.startIfEnabled()
  webUi = new WorkerWebUI(this, workDir, webUiPort)
  webUi.bind()
  workerWebUiUrl = s"http://$publicAddress:${webUi.boundPort}"
  registerWithMaster()

  metricsSystem.registerSource(workerSource)
  metricsSystem.start()
  metricsSystem.getServletHandlers.foreach(webUi.attachHandler)
}
```

根据代码清单 9-51，Worker 的 onStart 方法的执行步骤如下。

1）调用 createWorkDir 方法（见代码清单 9-52）创建 Worker 的工作目录。

2）如果配置了外部的 Shuffle 服务，那么启动 shuffleService。

3）创建 WorkerWebUI，并为其绑定端口。

4）拼接 Worker 的 WebUI 的 URL，并保存到 workerWebUiUrl 属性。

5）调用 registerWithMaster 方法（见代码清单 9-54）向 Master 注册 Worker。

6）将 workerSource 注册到 metricsSystem，然后启动 metricsSystem，最后将 metricsSystem 的 ServletContextHandler 添加到 WorkerWebUI。

代码清单9-52 创建Worker的工作目录

```
private def createWorkDir() {
  workDir = Option(workDirPath).map(new File(_)).getOrElse(new File(sparkHome,
      "work"))
  try {
    workDir.mkdirs()
    if ( !workDir.exists() || !workDir.isDirectory) {
      logError("Failed to create work directory " + workDir)
      System.exit(1)
    }
    assert (workDir.isDirectory)
  } catch {
    // 省略捕获到异常后退出进程的代码
}
```

2. 进程方式启动

Worker 的伴生对象中实现了 main 方法，这样就可以作为单独的 JVM 进程启动了。Main 函数的实现如代码清单 9-53 所示。

代码清单9-53 Worker伴生对象的main方法

```
def main(argStrings: Array[String]) {
  Utils.initDaemon(log)
  val conf = new SparkConf
  val args = new WorkerArguments(argStrings, conf)
  val rpcEnv = startRpcEnvAndEndpoint(args.host, args.port, args.webUiPort, args.cores,
    args.memory, args.masters, args.workDir, conf = conf)
  rpcEnv.awaitTermination()
}
```

根据代码清单 9-53，main 方法的执行步骤如下。

1）创建 SparkConf。

2）创建 WorkerArguments 以对执行 main 函数的参数进行解析，并将 Spark 属性配置文件中以 spark. 开头的属性保存到 SparkConf。

3）调用 Worker 伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-50）创建并启动 Worker 对象。

WorkerArguments 方法用于解析系统环境变量和启动 Worker 时指定的命令行参数，由于其实现和 MasterArguments 非常相似，因此对其源码不再展示。这里只列出 WorkerArguments 解析的参数，如表 9-3 所示。

表 9-3 WorkerArguments 解析的参数

属 性	含 义	默 认 值	系统环境变量	命令行参数
host	Worker 的监听地址	机器名	—	--ip 或者 -i（已弃用） --host 或者 -h
port	Worker 的监听端口	0	SPARK_WORKER_PORT	--port 或者 -p
webUiPort	WebUI 的监听端口	8081	SPARK_WORKER_WEBUI_PORT	--webui-port
propertiesFile	Spark 属性文件	spark-defaults.conf	—	--properties-file
cores	Worker 的内核数	系统内核数	SPARK_WORKER_CORES	--cores 或者 -c
memory	Worker 的内存大小	系统最大内存减 1GB	SPARK_WORKER_MEMORY	--memory 或者 -m
masters	Master 地址列表	null	—	例如，spark://abc,def
workDir	Worker 的工作目录	null	SPARK_WORKER_DIR	--work-dir 或者 -d



memory 默认为操作系统最大内存减去 1GB，这 1GB 是留给操作系统使用的。

WorkerArguments 的 parse 函数与 MasterArguments 中的 parse 函数的实现类似，读者

可自行阅读其源码。WorkerArguments 的 parse 函数要求必须在启动 Worker 时指定 spark:// 的参数作为 master 的 url。命令行参数的值会覆盖系统环境变量的值，解析的命令行参数如表 9-4 所示。

表 9-4 WorkerArguments 的 parse 函数解析的命令行参数

参 数 名	参 数 含 义
--ip 或者 -i	指定 host name，将来会停止使用，推荐使用 --host 或者 -h
--host 或者 -h	指定 host name
--port 或者 -p	指定端口
--webui-port	指定 WebUI 的端口
--properties-file	Spark 系统属性文件
--cores 或者 -c	指定内核数
--memory 或者 -m	指定内存大小
--work-dir 或者 -d	指定工作目录
--help	帮助


注意 命令行参数指定的值会覆盖系统环境变量指定的值。属性 spark.worker.ui.port 的值，会覆盖环境变量 SPARK_WORKER_WEBUI_PORT 和命令行参数 --webui-port 指定的值。

9.7.2 向 Master 注册 Worker

Worker 在启动后，需要加入到 Master 管理的整个集群中，以参与 Driver、Executor 的资源调度。Worker 要加入 Master 管理的集群，就必须将 Worker 注册到 Master。在启动 Worker 的过程中需要调用 registerWithMaster 方法向 Master 注册 Worker，其实现如代码清单 9-54 所示。

代码清单9-54 向Master注册Worker

```
private def registerWithMaster() {
  registrationRetryTimer match {
    case None =>
      registered = false
      registerMasterFutures = tryRegisterAllMasters() // 向所有的Master注册当前Worker
      connectionAttemptCount = 0
      registrationRetryTimer = Some(forwordMessageScheduler.scheduleAtFixedRate(
        new Runnable {
          override def run(): Unit = Utils.tryLogNonFatalError {
            Option(self).foreach(_.send(ReregisterWithMaster))
          }
        },
        INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS,
        INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS,
        TimeUnit.SECONDS))
```

```
      case Some(_) =>
        logInfo("Not spawning another attempt to register with the master, since there
            is an" +
          " attempt scheduled already.")
    }
  }
```

根据代码清单 9-54，registerWithMaster 方法的执行步骤如下。

1）调用 tryRegisterAllMasters 方法（见代码清单 9-55）向所有的 Master 注册当前 Worker。根据 9.6.5 节的内容，只有处于领导状态的 Master 来处理 Worker 的注册。

2）创建定时任务 registrationRetryTimer，按照常量 INITIAL_REGISTRATION_RETRY_INTERVAL_SECONDS 指定的间隔向 Worker 自身发送 ReregisterWithMaster 消息。在 Worker 的 receive 方法中使用了重载的 reregisterWithMaster 方法来处理 ReregisterWithMaster 消息，代码如下。

```
case ReregisterWithMaster =>
  reregisterWithMaster()
```

reregisterWithMaster 方法用于在 Worker 的 registered 为 false 时，重新向 Master 注册，由于其实现与 registerWithMaster 方法很多地方都是相似的，因此本文不对其进行介绍，留给感兴趣的读者自行查阅。

代码清单9-55 tryRegisterAllMasters的实现

```
private def tryRegisterAllMasters(): Array[JFuture[_]] = {
  masterRpcAddresses.map { masterAddress =>
    registerMasterThreadPool.submit(new Runnable {
      override def run(): Unit = {
        try {
          logInfo("Connecting to master " + masterAddress + "...")
          val masterEndpoint = rpcEnv.setupEndpointRef(masterAddress, Master.END-
              POINT_NAME)
          registerWithMaster(masterEndpoint)
        } catch {
          // 忽略异常处理代码
        }
      }
    })
  }
}
```

根据代码清单 9-55，tryRegisterAllMasters 方法实际上是遍历 masterRpcAddresses 中的每个 Master 的 RPC 地址，然后向 registerMasterThreadPool 提交向 Master 注册 Worker 的任务。此任务实际是调用重载的 registerWithMaster 方法（见代码清单 9-56）用于完成最后的注册。

代码清单9-56 重载的registerWithMaster方法

```
private def registerWithMaster(masterEndpoint: RpcEndpointRef): Unit = {
```

```
    masterEndpoint.ask[RegisterWorkerResponse](RegisterWorker(
      workerId, host, port, self, cores, memory, workerWebUiUrl))
      .onComplete {
        case Success(msg) =>
          Utils.tryLogNonFatalError {
            handleRegisterResponse(msg)
          }
        case Failure(e) =>
          logError(s"Cannot register with master: ${masterEndpoint.address}", e)
          System.exit(1)
      }(ThreadUtils.sameThread)
  }
```

根据代码清单 9-56，registerWithMaster 方法主要是向 Master 发送 RegisterWorker 消息（此消息将携带 Worker 的 ID、host、port、内核数、内存大小等信息），并对返回的结果使用 handleRegisterResponse 方法处理。Master 接收 RegisterWorker 消息的处理过程已在 9.6.5 节介绍过了，这里主要分析 handleRegisterResponse 方法的实现，如代码清单 9-57 所示。

代码清单9-57　handleRegisterResponse的实现

```
  private def handleRegisterResponse(msg: RegisterWorkerResponse): Unit = synchronized {
    msg match {
      case RegisteredWorker(masterRef, masterWebUiUrl) =>
        logInfo("Successfully registered with master " + masterRef.address.toSparkURL)
        registered = true
        changeMaster(masterRef, masterWebUiUrl) // 切换Master
        forwordMessageScheduler.scheduleAtFixedRate(new Runnable {
          override def run(): Unit = Utils.tryLogNonFatalError {
            self.send(SendHeartbeat) // 定时向Master发送心跳
          }
        }, 0, HEARTBEAT_MILLIS, TimeUnit.MILLISECONDS)
        if (CLEANUP_ENABLED) {
          logInfo(
            s"Worker cleanup enabled; old application directories will be deleted in:
                $workDir")
          forwordMessageScheduler.scheduleAtFixedRate(new Runnable {
            override def run(): Unit = Utils.tryLogNonFatalError {
              self.send(WorkDirCleanup) // 定时清理工作目录
            }
          }, CLEANUP_INTERVAL_MILLIS, CLEANUP_INTERVAL_MILLIS, TimeUnit.MILLISECONDS)
        }

        val execs = executors.values.map { e =>
          new ExecutorDescription(e.appId, e.execId, e.cores, e.state)
        }
        masterRef.send(WorkerLatestState(workerId, execs.toList, drivers.keys.
            toSeq)) //汇报状态

      case RegisterWorkerFailed(message) =>
        if (!registered) {
          logError("Worker registration failed: " + message)
          System.exit(1)
        }
```

```
      case MasterInStandby => // Master还未准备好或者Master处于支持（STANDBY）状态
    }
  }
```

根据9.6.5节对Master注册Worker的分析，我们知道Master将给Worker回复三种可能的消息，而handleRegisterResponse方法也的确处理了这三种消息。

1）处理RegisteredWorker消息。如果Master回复了此消息，说明Worker已经成功在Master上注册，具体的处理步骤如下。

① 将registered设置为true。

② 调用changeMaster方法（见代码清单9-58）修改激活的Master的信息。changeMaster方法还调用cancelLastRegistrationRetry方法取消注册尝试（即取消了调用tryRegisterAllMasters方法产生的注册任务和registrationRetryTimer）。

③ 向forwordMessageScheduler提交以HEARTBEAT_MILLIS作为间隔向Worker自身发送SendHeartbeat消息的定时任务。9.7.3节对Worker处理SendHeartbeat消息的过程进行了详细的分析。

④ 如果允许清理工作目录（即CLEANUP_ENABLED为true），向forwordMessageScheduler提交以CLEANUP_INTERVAL_MILLIS作为间隔向Worker自身发送WorkDirCleanup消息的定时任务。

⑤ 向Master发送WorkerLatestState消息（此消息携带Worker的身份标识、Worker节点的所有Executor的描述信息、调度到当前Worker的所有Driver的身份标识）。

代码清单9-58 changeMaster方法的实现

```
private def changeMaster(masterRef: RpcEndpointRef, uiUrl: String) {
  activeMasterUrl = masterRef.address.toSparkURL
  activeMasterWebUiUrl = uiUrl
  master = Some(masterRef)
  connected = true
  if (conf.getBoolean("spark.ui.reverseProxy", false)) {
    logInfo(s"WorkerWebUI is available at $activeMasterWebUiUrl/proxy/$workerId")
  }
  cancelLastRegistrationRetry()
}

private def cancelLastRegistrationRetry(): Unit = {
  if (registerMasterFutures != null) {
    registerMasterFutures.foreach(_.cancel(true))
    registerMasterFutures = null
  }
  registrationRetryTimer.foreach(_.cancel(true))
  registrationRetryTimer = None
}
```

2）处理RegisterWorkerFailed消息。如果Master回复了此消息，说明Worker在Master上注册失败了。如果Worker还未向任何Master节点注册成功，那么退出Worker进程。

3）处理 MasterInStandby 消息。如果 Master 回复了此消息，说明 Master 处于 Standby 的状态，并不是领导身份，此时 Worker 不作任何处理。

有了对 Master 和 Worker 各自有关注册 Worker 的实现分析，我们用图 9-16 来表示注册 Worker 的完整流程。

这里对图 9-16 中的各个序号进行说明。

序号①：表示 Worker 刚刚启动时，由于不知道哪个 Master 是 ALIVE 的，所以向所有 Master 发送 RegisterWorker 消息（此消息将携带 Worker 的 ID、host、port、内核数、内存大小等信息）。向状态为 STANDBY 的 Master 发送消息将接收到 MasterInStandby 消息，Worker 由此知道这个 Master 不“管事”。如果 Worker 向 ALIVE 状态的 Master 重复注册或者注册失败，将会收到 RegisterWorkerFailed 消息，Worker 发现自己还未注册成功时将退出。

序号②：状态为 ALIVE 的 Master 接收到 RegisterWorker 消息后，将根据 RegisterWorker 消息携带的信息构建出 WorkerInfo，并将 WorkerInfo 添加到 workers、idToWorker、addressToWorker 等缓存中，最后调用 PersistenceEngine 的 addWorker 方法将 WorkerInfo 持久化。

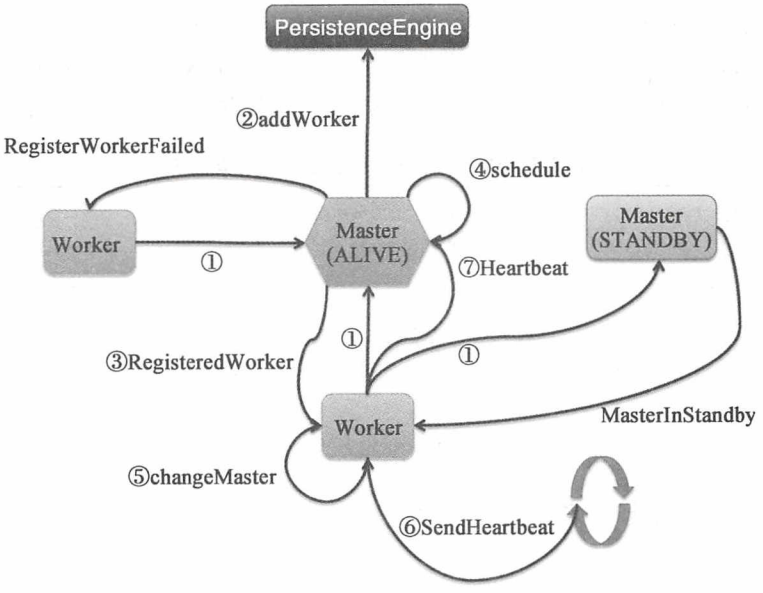


图 9-16　注册 Worker 的完整流程

序号③：状态为 ALIVE 的 Master 在 Worker 注册成功后，向 Worker 回复 RegisteredWorker 消息。

序号④：状态为 ALIVE 的 Master 在向 Worker 回复 RegisteredWorker 消息后，将调用

schedule 对 Driver 及 Executor 进行资源调度。

序号⑤：Worker 接收到 Master 回复的 RegisteredWorker 消息后，将调用 changeMaster 方法修改激活的 Master 的信息。

序号⑥：Worker 调用 changeMaster 方法后，将向 forwordMessageScheduler 提交向 Worker 自身发送 SendHeartbeat 消息的定时任务。

序号⑦：Worker 接收到 SendHeartbeat 消息后，将向 Master 发送心跳（Heartbeat）消息。Master 接收到 Heartbeat 消息后，将更新 WorkerInfo 的最后心跳时间（lastHeartbeat）。

9.7.3 向 Master 发送心跳

为了让 Master 得知 Worker 依然健康运行着，就需要不断地告诉 Master：“我活着”，这个过程是通过发送心跳实现的。

根据之前的内容我们知道，当 Worker 向 Master 注册成功后会接收到 Master 回复的 RegisteredWorker 消息，Worker 使用 handleRegisterResponse 方法处理 RegisteredWorker 消息时，将会向 forwordMessageScheduler 提交以 HEARTBEAT_MILLIS 作为间隔向 Worker 自身发送 SendHeartbeat 消息的定时任务。Worker 的 receive 方法实现了对 SendHeartbeat 消息的处理，代码如下。

```
case SendHeartbeat =>
  if (connected) { sendToMaster(Heartbeat(workerId, self)) }
```

根据上述代码，我们知道如果 connected 为 true，则调用 sendToMaster 方法（见代码清单 9-59），向 Master 发送 Heartbeat 消息（此消息将携带 Worker 的 ID 和 Worker 自己的 RpcEndpointRef）。

代码清单9-59 sendToMaster方法的实现

```
private def sendToMaster(message: Any): Unit = {
  master match {
    case Some(masterRef) => masterRef.send(message)
    case None =>
      // 忽略Worker还未与Master建立连接时不作任何处理，只打印日志的代码
  }
}
```

根据 9.6.7 节的内容我们知道，Master 在处理 Heartbeat 消息时有可能向 Worker 发送 ReconnectWorker 消息，Worker 的 receive 方法接收到 ReconnectWorker 消息后只是再次调用了 registerWithMaster 方法（见代码清单 9-54）向 Master 注册 Worker，代码如下。

```
case ReconnectWorker(masterUrl) =>
  registerWithMaster()
```

经过对 Master 中检查 Worker 超时（9.6.2 节）、Master 处理 Worker 心跳（9.6.7 节），以及本节对 Worker 向 Master 发送心跳等环节的分析，我们可以用图 9-17 来表示 Worker 心跳

及超时检查的流程。

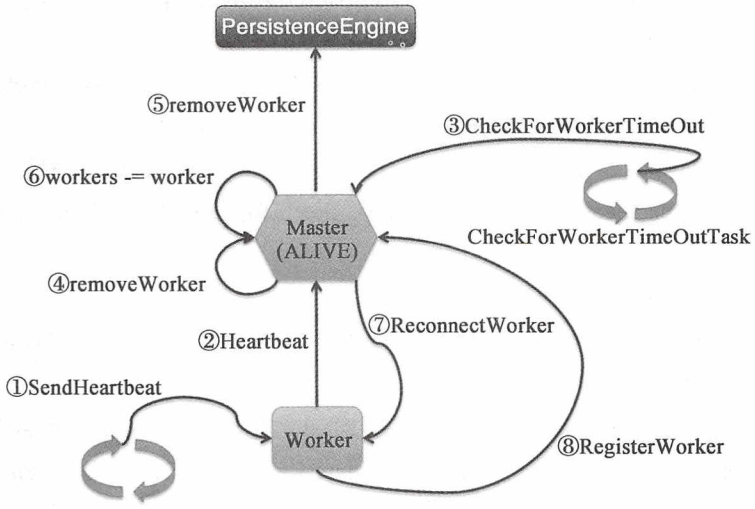


图 9-17　Worker 心跳及超时检查的流程

这里对图 9-17 中的各个序号进行说明。

序号①：Worker 在向 Master 注册成功后，将向 forwordMessageScheduler 提交以 HEARTBEAT_MILLIS 作为间隔向 Worker 自身发送 SendHeartbeat 消息的定时任务。

序号②：Worker 接收到 SendHeartbeat 消息后，将向 Master 发送 Heartbeat 消息。在正常情况下，Worker 向 Master 发送的心跳足够及时，因此 Master 并未检查出 Worker 超时，此时 idToWorker 中将缓存着 WorkerInfo，Master 只需要将此 WorkerInfo 的最后心跳时间（lastHeartbeat）更新为系统当前时间戳。

序号③：Master 内部的定时任务 checkForWorkerTimeOutTask 将以 WORKER_TIMEOUT_MS 为时间间隔定时向 Master 自身发送 CheckForWorkerTimeOut 消息。Master 接收到 CheckForWorkerTimeOut 消息后，调用 timeOutDeadWorkers 方法检查超时的 Worker。

序号④：Master 在检查过程中发现 Worker 超时，但是对应的 WorkerInfo 的状态不是 DEAD，因此调用 removeWorker 方法移除 Master 维护的关于指定 Worker 的相关信息和状态。这里特别要关注的是，此时 Master 会将 WorkerInfo 从 idToWorker 中移除，但是 workers 中仍然保留 WorkerInfo。

序号⑤：removeWorker 方法的最后还将调用持久化引擎的 removeWorker 方法从持久化存储中移除 WorkerInfo。

序号⑥：Master 在检查过程中发现 Worker 超时且对应的 WorkerInfo 的状态是 DEAD，等待足够长的时间后将 WorkerInfo 从 workers 中移除。

序号⑦：Master 在检查 Worker 时，如果发生了序号④所示的情况，那么 idToWorker 中将没有对应的 WorkerInfo，但是 workers 中仍然保留 WorkerInfo，此时 Master 将向

Worker 发送 ReconnectWorker 消息。

序号⑧：Worker 接收到 ReconnectWorker 消息后，将重新调用 registerWithMaster 方法向 Master 注册，进而向 Master 发送 RegisterWorker 消息。

9.7.4 Worker 与领导选举

在 9.6.3 节介绍 Master 被选举为领导时，将会向 Worker 发送 MasterChanged 消息。Worker 的 receive 方法接收到 MasterChanged 消息后的处理如代码清单 9-60 所示。

代码清单9-60 Worker对MasterChanged消息的处理

```
case MasterChanged(masterRef, masterWebUiUrl) =>
  logInfo("Master has changed, new master is at " + masterRef.address.toSparkURL)
  changeMaster(masterRef, masterWebUiUrl)

  val execs = executors.values.
    map(e => new ExecutorDescription(e.appId, e.execId, e.cores, e.state))
  masterRef.send(WorkerSchedulerStateResponse(workerId, execs.toList, drivers.keys.toSeq))
```

根据代码清单 9-60，Worker 接收到 MasterChanged 消息后的处理步骤如下。

1）调用 changeMaster 方法（见代码清单 9-58）改变激活的 Master 的相关信息。

2）通过遍历 executors 中的每个 ExecutorRunner，利用 ExecutorRunner 的信息创建 ExecutorDescription。

3）向 Master 发送 WorkerSchedulerStateResponse 消息。WorkerSchedulerStateResponse 消息携带着 Worker 的 ID、ExecutorDescription 列表、所有 Driver 的 ID 等信息。Master 接收到 WorkerSchedulerStateResponse 消息后将更新 Worker 的调度状态。由于 Master 处理 WorkerSchedulerStateResponse 消息的实现非常简单，因此留给感兴趣的读者自行阅读。

经过对持久化引擎、领导选举代理、Master 被选举为领导时的处理（9.6.3 节）、Master 如何注册 Worker、Master 如何注册 Aplication，以及本节对 Worker 处理 MasterChanged 消息等环节的分析，我们可以用图 9-18 来表示 Master 领导选举的整个流程。

图 9-18 中显示了三个 Master，其中 Master 0 是 ALIVE 状态的，Master 1 和 Master 2 是 STANDBY 状态的。图 9-18 中的各个序号的含义如下。

序号①：Worker 在启动后会向 Master 注册，Driver 在启动后也会向 Master 注册。在注册成功后，Master 会调用 PersistenceEngine 的 addWorker、addApplication 及 addDriver 三个方法，将 Worker、Driver、Application 的相关信息持久化。

序号②：Master 在运行的时候出现网络故障、宕机或者服务下线时，ZooKeeper 会感知到。

序号③：Master 1 和 Master 2 参加领导选举，最后 Master 1 胜出，于是 LeaderLatch 将调用 Master 1 的 ZooKeeperLeaderElectionAgent 的 isLeader 方法，isLeader 方法实际调用了 Master 1 的 electedLeader 方法。

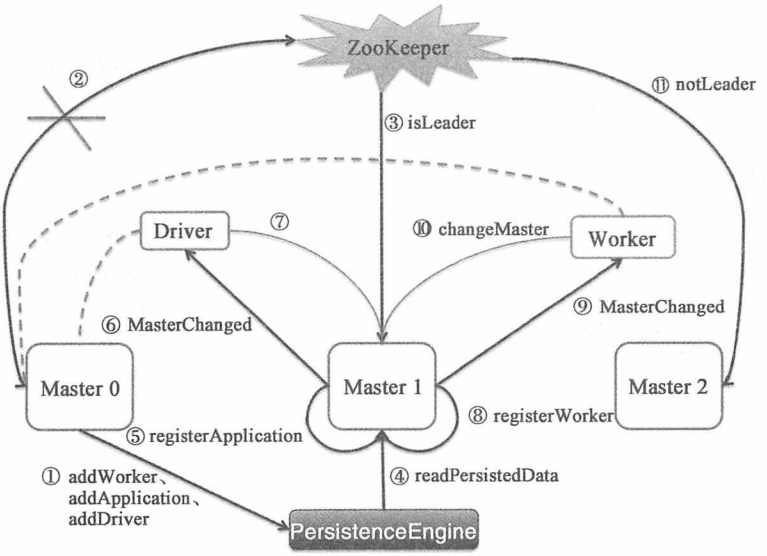


图 9-18　Master 领导选举的流程

序号④：electedLeader 方法调用 PersistenceEngine 的 readPersistedData 方法从持久化数据中读取 WorkerInfo、DriverInfo、ApplicationInfo，然后调用 Master 1 的 beginRecovery 方法开始恢复。

序号⑤：beginRecovery 方法对于从持久化引擎中读取的 ApplicationInfo，调用 registerApplication 方法注册到 apps、idToApp、endpointToApp、addressToApp、waitingApps 等缓存中，然后将 ApplicationInfo 的状态设置为 UNKNOWN。

序号⑥：beginRecovery 方法还将向提交 Application 的 Driver 发送 MasterChanged 消息。

序号⑦：Driver 接收到 MasterChanged 消息后，将自身的 master 属性修改为当前 Master 的 RpcEndpointRef，并将 alreadyDisconnected 设置为 false，最后向 Master 发送 MasterChangeAcknowledged 消息。Master 接收到 MasterChangeAcknowledged 消息后，将 ApplicationInfo 的状态修改为 WAITING，然后在不存在状态为 UNKNOWN 的 ApplicationInfo 和 WorkerInfo 时调用 completeRecovery 方法（见代码清单 9-32）完成恢复。

序号⑧：beginRecovery 方法对于从持久化引擎中读取的 WorkerInfo，调用 registerWorker 方法将 WorkerInfo 添加到 workers、idToWorker、addressToWorker 等缓存中，然后将 WorkerInfo 的状态设置为 UNKNOWN。

序号⑨：beginRecovery 方法还将向 Worker 发送 MasterChanged 消息。

序号⑩：Worker 接收到 MasterChanged 消息后，将自身的 activeMasterUrl、activeMasterWebUiUrl、master 等属性修改为当前 Master 的对应信息，然后将 connected 设置为 true，最后向 Master 发送 WorkerSchedulerStateResponse 消息。Master 接收到 WorkerSchedulerStateResponse 消息后，首先将 WorkerInfo 的状态修改为 ALIVE，然后对此 Worker 上的

Executor 和 Driver 也进行恢复，最后在不存在状态为 UNKNOWN 的 ApplicationInfo 和 WorkerInfo 时调用 completeRecovery 方法（见代码清单 9-32）完成恢复。

序号⑪：Master 1 和 Master 2 参加领导选举，最后 Master 1 胜出，于是 LeaderLatch 将调用 Master 2 的 ZooKeeperLeaderElectionAgent 的 notLeader 方法，notLeader 方法实际调用了 Master 2 的 revokedLeadership 方法。

9.7.5 运行 Driver

在介绍 Master 对 Driver 的资源调度和运行时，我们知道 Master 将向 Worker 发送 LaunchDriver 消息以运行 Driver。下面一起来看 Worker 接收到 LaunchDriver 消息后，是如何运行 Driver 的。

Worker 的 receive 方法中实现了对 LaunchDriver 消息的处理，代码如下。

```
case LaunchDriver(driverId, driverDesc) =>
  val driver = new DriverRunner(conf, driverId, workDir, sparkHome,
    driverDesc.copy(command = Worker.maybeUpdateSSLSettings(driverDesc.command,
        conf)),
    self, workerUri, securityMgr)
  drivers(driverId) = driver
  driver.start()

  coresUsed += driverDesc.cores
  memoryUsed += driverDesc.mem
```

根据上述代码，我们知道 Worker 处理 LaunchDriver 消息的步骤如下。

1）创建 DriverRunner。

2）将 Driver 的身份标识与 DriverRunner 的关系放入 drivers 中缓存。

3）调用 DriverRunner 的 start 方法（见代码清单 9-61）启动 DriverRunner。

4）更新 Worker 已经使用的内核数和内存大小。

代码清单9-61 DriverRunner的start方法

```
private[worker] def start() = {
  new Thread("DriverRunner for " + driverId) {
    override def run() {
      var shutdownHook: AnyRef = null
      try {
        // 省略关闭钩子的代码，关闭钩子用于在JVM进程退出前“杀死”Driver进程
        val exitCode = prepareAndRunDriver()

        finalState = if (exitCode == 0) {
          Some(DriverState.FINISHED)
        } else if (killed) {
          Some(DriverState.KILLED)
        } else {
          Some(DriverState.FAILED)
        }
      } catch {
```

```
          // 省略异常处理的代码，发生异常时将“杀死”Driver进程
        } finally {
          // 省略移除钩子的代码
        }
        worker.send(DriverStateChanged(driverId, finalState.get, finalException))
      }
    }.start()
  }
```

根据代码清单 9-61，DriverRunner 的 start 方法的核心在于 prepareAndRunDriver 方法。prepareAndRunDriver 方法的实现如代码清单 9-62 所示。

代码清单9-62 prepareAndRunDriver方法的实现

```
private[worker] def prepareAndRunDriver(): Int = {
  val driverDir = createWorkingDirectory()
  val localJarFilename = downloadUserJar(driverDir)

  def substituteVariables(argument: String): String = argument match {
    case "{{WORKER_URL}}" => workerUrl
    case "{{USER_JAR}}" => localJarFilename
    case other => other
  }
  val builder = CommandUtils.buildProcessBuilder(driverDesc.command, securityManager,
    driverDesc.mem, sparkHome.getAbsolutePath, substituteVariables)

  runDriver(builder, driverDir, driverDesc.supervise)
}
```

根据代码清单 9-62，prepareAndRunDriver 方法的执行步骤如下。

1）创建 Driver 的工作目录。

2）下载用户指定的 Jar 文件。

3）构造 ProcessBuilder，进程命令将由 DriverDescription 的 command 属性决定。ProcessBuilder 执行 java 命令的主类是 org.apache.spark.deploy.worker.DriverWrapper，有关 buildProcessBuilder 的实现，请参阅附录 F。在调用 CommandUtils 的 buildProcessBuilder 方法时将 substituteVariables 方法作为了参数，substituteVariables 方法的作用是将 DriverDescription 的 command 属性（即 Command）的 arguments 属性中每个参数中的变量替换为指定的值。例如，将 {WORKER_URL} 替换为 DriverRunner 的 workerUrl 属性。

4）调用 runDriver 方法运行 Driver。由于在 Standalone 集群中运行 Driver 的方式已经被抛弃，并会在将来从 Spark 中移除，所以对运行 Driver 的分析就讲到这里。对 runDriver 方法感兴趣的读者可以自行研究。

9.7.6 运行 Executor

在 9.6.4 节介绍 Master 的资源调度时，我们知道 Master 向 Worker 发送 LaunchExecutor 消息以运行 Executor。LaunchExecutor 消息携带着 masterUrl、Application 的 ID、Executor

的ID、应用的描述信息、Executor分配获得的内核数、Executor分配获得的内存大小等信息。Worker在接收到LaunchExecutor消息后的处理如下。

```
case LaunchExecutor(masterUrl, appId, execId, appDesc, cores_, memory_) =>
  if (masterUrl != activeMasterUrl) {
    logWarning("Invalid Master (" + masterUrl + ") attempted to launch executor.")
  } else {
    try {
      logInfo("Asked to launch executor %s/%d for %s".format(appId, execId, appDesc.
          name))

      val executorDir = new File(workDir, appId + "/" + execId)
      if (!executorDir.mkdirs()) {
        throw new IOException("Failed to create directory " + executorDir)
      }

      val appLocalDirs = appDirectories.getOrElse(appId,
        Utils.getOrCreateLocalRootDirs(conf).map { dir =>
          val appDir = Utils.createDirectory(dir, namePrefix = "executor")
          Utils.chmod700(appDir)
          appDir.getAbsolutePath()
        }.toSeq)
      appDirectories(appId) = appLocalDirs
      val manager = new ExecutorRunner(appId, execId,
        appDesc.copy(command = Worker.maybeUpdateSSLSettings(appDesc.command,
            conf)),
        cores_, memory_, self, workerId, host, webUi.boundPort, publicAddress,
            sparkHome,
        executorDir, workerUri, conf, appLocalDirs, ExecutorState.RUNNING)
      executors(appId + "/" + execId) = manager
      manager.start()
      coresUsed += cores_
      memoryUsed += memory_
      sendToMaster(ExecutorStateChanged(appId, execId, manager.state, None,
          None))
    } catch {
      case e: Exception =>
        logError(s"Failed to launch executor $appId/$execId for ${appDesc.
            name}.", e)
        if (executors.contains(appId + "/" + execId)) {
          executors(appId + "/" + execId).kill()
          executors -= appId + "/" + execId
        }
        sendToMaster(ExecutorStateChanged(appId, execId, ExecutorState.FAILED,
          Some(e.toString), None))
    }
  }
```

根据上述代码，Worker接收到LaunchExecutor消息后，首先判断发送LaunchExecutor消息的Master是否是激活的Master，如果不是，则什么都不做，否则进行以下处理。

1）在Worker的工作目录下创建Executor的工作目录。

2）给Executor创建本地目录，并将这些目录缓存到Worker的appDirectories属性中。这些目录将通过环境变量SPARK_EXECUTOR_DIRS传递给Executor，并在Application完

成后由 Worker 删除。

3）创建 ExecutorRunner，并将 ExecutorRunner 放入 executors。例如，假设 appId 为 app-20170528125247-0000，execId 为 0，那么将 app-20170528125247-0000/0 与 ExecutorRunner 的映射关系放入了 executors。在创建完 ExecutorRunner 时，ExecutorRunner 的状态为 RUNNING。

4）调用 ExecutorRunner 的 start 方法（见代码清单 9-63）启动 ExecutorRunner。

5）更新已经使用的内核数 coresUsed 和已经使用的内存大小 memoryUsed。

6）向 Master 发送 ExecutorStateChanged 消息以更新 Executor 的状态。

代码清单9-63 ExecutorRunner的start方法

```
private[worker] def start() {
  workerThread = new Thread("ExecutorRunner for " + fullId) {
    override def run() { fetchAndRunExecutor() }
  }
  workerThread.start()
  // 省略关闭钩子的代码，关闭钩子用于在JVM进程退出前“杀死”Executor进程
}
```

根据代码清单 9-63，ExecutorRunner 的 start 方法的核心在于 workerThread，workerThread 在运行时将执行 fetchAndRunExecutor 方法。fetchAndRunExecutor 方法的实现如代码清单 9-64 所示。

代码清单9-64 fetchAndRunExecutor方法的实现

```
private def fetchAndRunExecutor() {
  try {
    val builder = CommandUtils.buildProcessBuilder(appDesc.command, new Security-
       Manager(conf),
      memory, sparkHome.getAbsolutePath, substituteVariables) // 构造ProcessBuilder
    val command = builder.command()
    val formattedCommand = command.asScala.mkString("\"", "\" \"", "\"")
    logInfo(s"Launch command: $formattedCommand")
    // 为ProcessBuilder设置执行目录和环境变量
    builder.directory(executorDir)
    builder.environment.put("SPARK_EXECUTOR_DIRS", appLocalDirs.mkString(File.
       pathSeparator))
    builder.environment.put("SPARK_LAUNCH_WITH_SCALA", "0")
    val baseUrl =
      if (conf.getBoolean("spark.ui.reverseProxy", false)) {
        s"/proxy/$workerId/logPage/?appId=$appId&executorId=$execId&logType="
      } else {
        s"http://$publicAddress:$webUiPort/logPage/?appId=$appId&executorId=$exe
           cId&logType="
      }
    builder.environment.put("SPARK_LOG_URL_STDERR", s"${baseUrl}stderr")
    builder.environment.put("SPARK_LOG_URL_STDOUT", s"${baseUrl}stdout")

    process = builder.start() // 启动ProcessBuilder，生成进程
    val header = "Spark Executor Command: %s\n%s\n\n".format(
```

```
        formattedCommand, "=" * 40)

      val stdout = new File(executorDir, "stdout") // 重定向进程的输出流与错误流
      stdoutAppender = FileAppender(process.getInputStream, stdout, conf)
      val stderr = new File(executorDir, "stderr")
      Files.write(header, stderr, StandardCharsets.UTF_8)
      stderrAppender = FileAppender(process.getErrorStream, stderr, conf)
      val exitCode = process.waitFor() // 等待获取进程的退出状态
      state = ExecutorState.EXITED
      val message = "Command exited with code " + exitCode
      worker.send(ExecutorStateChanged(appId, execId, state, Some(message), Some
          (exitCode)))
    } catch {
      case interrupted: InterruptedException =>
        logInfo("Runner thread for executor " + fullId + " interrupted")
        state = ExecutorState.KILLED
        killProcess(None)
      case e: Exception =>
        logError("Error running executor", e)
        state = ExecutorState.FAILED
        killProcess(Some(e.toString))
    }
  }
```

根据代码清单 9-64，fetchAndRunExecutor 方法的执行步骤如下。

1）构造 ProcessBuilder，进程命令将由 ApplicationDescription 的 command 属性决定。ApplicationDescription 的 command 属性是何时产生的呢？本书将在 9.9.3 节介绍。进程命令的主类是 org.apache.spark.executor.CoarseGrainedExecutorBackend（将在 9.10 节详细介绍其实现），有关 buildProcessBuilder 的实现，请参阅附录 F。在调用 CommandUtils 的 buildProcessBuilder 方法时将 substituteVariables 方法作为了参数，substituteVariables 方法（见代码清单 9-65）的作用是将 ApplicationDescription 的 command 属性（即 Command）的 arguments 属性中每个参数中的变量替换为指定的值。例如，将 {WORKER_URL} 替换为 ExecutorRunner 的 workerUrl 属性。

2）将 executorDir 设置为 ProcessBuilder 执行目录。这里的 executorDir 就是 Worker 在接收到 LaunchExecutor 消息后创建的那个 executorDir。

3）为 ProcessBuilder 设置环境变量。

4）启动 ProcessBuilder，生成进程。一旦执行了 builder.start()，我们利用 jdk 自带的工具 VisualVM，就会发现此时已经产生了一个新的进程，笔者对本地的 CoarseGrained-ExecutorBackend 进程的截图如图 9-19 所示。CoarseGrainedExecutorBackend 进程的系统属性如图 9-20 所示。

5）重定向进程的输出流与错误流为 executorDir 目录下的文件 stdout 与 stderr。以笔者本地为例，分别为 D:\git\spark\work\app-20150707144151-0000\0\stdout 和 D:\git\spark\work\ app-20150707144151-0000\0\stderr。

6）等待获取进程的退出状态，一旦收到退出状态，则向 Worker 发送 Executor-

StateChanged 消息。



图 9-19 CoarseGrainedExecutorBackend 进程的截图

代码清单9-65 替换命令中的变量

```
private[worker] def substituteVariables(argument:
    String): String = argument match {
  case "{{WORKER_URL}}" => workerUrl
  case "{{EXECUTOR_ID}}" => execId.toString
  case "{{HOSTNAME}}" => host
  case "{{CORES}}" => cores.toString
  case "{{APP_ID}}" => appId
  case other => other
}
```

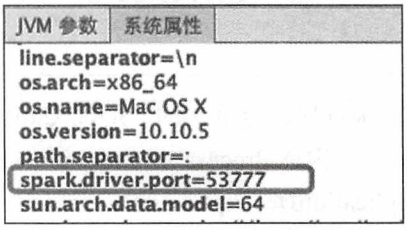


图 9-20 CoarseGrainedExecutorBackend 进程的系统属性

根据代码清单 9-64，如果在执行 fetchAndRunExecutor 方法的过程中捕获到异常，将调用 killProcess 方法。killProcess 方法的实现如代码清单 9-66 所示。

根据代码清单 9-66，killProcess 方法在“杀死”CoarseGrainedExecutorBackend 进程后将向 Worker 发送 ExecutorStateChanged 消息，有关 ExecutorStateChanged 消息的处理将在下一小节介绍。

代码清单9-66 “杀死”CoarseGrainedExecutorBackend进程

```
private def killProcess(message: Option[String]) {
  var exitCode: Option[Int] = None
  if (process != null) {
    logInfo("Killing process!")
    if (stdoutAppender != null) {
      stdoutAppender.stop()
    }
    if (stderrAppender != null) {
      stderrAppender.stop()
    }
    exitCode = Utils.terminateProcess(process, EXECUTOR_TERMINATE_TIMEOUT_MS)
    if (exitCode.isEmpty) {
```

```
        logWarning("Failed to terminate process: " + process +
          ". This process will likely be orphaned.")
      }
    }
    try {
      worker.send(ExecutorStateChanged(appId, execId, state, message, exitCode))
    } catch {
      case e: IllegalStateException => logWarning(e.getMessage(), e)
    }
  }
```

9.7.7 处理 Executor 的状态变化

Worker 接收 ExecutorStateChanged 消息，以对 Executor 的状态发生变化后进行处理。Worker 处理 ExecutorStateChanged 消息的代码如下。

```
    case executorStateChanged @ ExecutorStateChanged(appId, execId, state, message,
        exitStatus) =>
  handleExecutorStateChanged(executorStateChanged)
```

根据上述代码，Worker 将调用 handleExecutorStateChanged 方法处理 Executor 的状态变化。handleExecutorStateChanged 方法的实现如代码清单 9-67 所示。

代码清单9-67 处理ExecutorStateChanged消息

```
  private[worker] def handleExecutorStateChanged(executorStateChanged: Executor-
      StateChanged):
    Unit = {
    sendToMaster(executorStateChanged)
    val state = executorStateChanged.state
    if (ExecutorState.isFinished(state)) {
      val appId = executorStateChanged.appId
      val fullId = appId + "/" + executorStateChanged.execId
      val message = executorStateChanged.message
      val exitStatus = executorStateChanged.exitStatus
      executors.get(fullId) match {
        case Some(executor) =>
          logInfo("Executor " + fullId + " finished with state " + state +
            message.map(" message " + _).getOrElse("") +
            exitStatus.map(" exitStatus " + _).getOrElse(""))
          executors -= fullId
          finishedExecutors(fullId) = executor
          trimFinishedExecutorsIfNecessary()
          coresUsed -= executor.cores
          memoryUsed -= executor.memory
        case None =>
          logInfo("Unknown Executor " + fullId + " finished with state " + state +
            message.map(" message " + _).getOrElse("") +
            exitStatus.map(" exitStatus " + _).getOrElse(""))
      }
      maybeCleanupApplication(appId)
    }
  }
```

根据代码清单 9-67，handleExecutorStateChanged 方法除了更新 Worker 中与此 Executor 有关的缓存信息外，还向 Master 转发了 ExecutorStateChanged 消息。Master 对 ExecutorStateChanged 消息的处理已在 9.6.10 节详细介绍过。

9.8 StandaloneAppClient 实现

StandaloneAppClient 是在 Standalone 模式下，Application 与集群管理器进行对话的客户端。深入分析 StandaloneAppClient 之前，我们还是先来了解它的属性。

- rpcEnv：即 SparkContext 的 SparkEnv 的 RpcEnv。
- masterUrls：用于缓存每个 Master 的 spark://host:port 格式的 URL 的数组。
- appDescription：Application 的描述信息（ApplicationDescription）。ApplicationDescription 中记录了 Application 的名称（name）、Application 需要的最大内核数（maxCores）、每个 Executor 所需要的内存大小（memoryPerExecutorMB）、运行 CoarseGrainedExecutorBackend 进程的命令（command）、Spark UI 的 URL（appUiUrl）、事件日志的路径（eventLogDir）、事件日志采用的压缩算法名（eventLogCodec）、每个 Executor 所需的内核数（coresPerExecutor）、提交 Application 的用户名（user）等信息。
- listener：类型为 StandaloneAppClientListener，是对集群事件的监听器。StandaloneAppClientListener 有两个实现类，分别是 StandaloneSchedulerBackend 和 AppClientCollector。AppClientCollector 只用于测试，StandaloneSchedulerBackend 可用在 local-cluster 或 Standalone 模式下。StandaloneSchedulerBackend 在构造 StandaloneAppClient 时会将自身作为 listener 参数。
- conf：即 SparkConf。
- masterRpcAddresses：Master 的 RPC 地址（RpcAddress）。
- REGISTRATION_TIMEOUT_SECONDS：注册超时的秒数，固定为 20。
- REGISTRATION_RETRIES：注册的重试次数，固定为 3。
- endpoint：类型为 AtomicReference，用于持有 ClientEndpoint 的 RpcEndpointRef。本书在 9.8.1 节将详细分析 ClientEndpoint 的实现。


注意 此处所介绍的 ClientEndpoint 是 org.apache.spark.deploy.client.ClientEndpoint，而不是 org.apache.spark.deploy.ClientEndpoint。

- appId：类型为 AtomicReference，用于持有 Application 的 ID。
- registered：类型为 AtomicReference，用于标识是否已经将 Application 注册到 Master。

有了对 StandaloneAppClient 中包含属性的了解，下面首先分析 ClientEndpoint，然后介绍 StandaloneAppClient 的功能。

9.8.1 ClientEndpoint的实现分析

我们刚才说StandaloneAppClient是在Standalone模式下，Application与集群进行对话的客户端，而Spark各个组件之间的通信离不开RpcEnv及RpcEndpoint。ClientEndpoint继承自ThreadSafeRpcEndpoint，也是StandaloneAppClient的内部类，StandaloneAppClient依赖于ClientEndpoint与集群管理器进行通信。

ClientEndpoint的属性如下。

- rpcEnv：即SparkContext的SparkEnv的RpcEnv。
- master：处于激活状态的Master的RpcEndpointRef。
- alreadyDisconnected：是否已经与Master断开连接。此属性用于防止多次调用StandaloneAppClientListener的disconnected方法。
- alreadyDead：表示ClientEndpoint是否已经“死掉”。此属性用于防止多次调用StandaloneAppClientListener的dead方法。
- registerMasterThreadPool：用于向Master注册Application的ThreadPoolExecutor。registerMasterThreadPool的线程池大小为外部类StandaloneAppClient的masterRpcAddresses数组的大小，启动的线程以appclient-register-master-threadpool为前缀。
- registerMasterFutures：用于保存registerMasterThreadPool执行的向各个Master注册Application的任务返回的Future。
- registrationRetryThread：类型为ScheduledExecutorService，用于对Application向Master进行注册的重试。由registrationRetryThread启动的线程名称为appclient-registration-retry-thread。
- registrationRetryTimer：用于持有向registrationRetryThread提交关于注册的定时调度返回的ScheduledFuture。

了解了ClientEndpoint中包含的属性，现在来看看ClientEndpoint中的重要功能。

1. ClientEndpoint的启动

将ClientEndpoint注册到RpcEnv的Dispatcher时，会触发对ClientEndpoint的onStart方法的调用。ClientEndpoint的onStart方法的实现如代码清单9-68所示。

代码清单9-68 ClientEndpoint的onStart方法

```
override def onStart(): Unit = {
  try {
    registerWithMaster(1)
  } catch {
    case e: Exception =>
      logWarning("Failed to connect to master", e)
      markDisconnected()
      stop()
  }
}
```

根据代码清单 9-68，ClientEndpoint 的 onStart 方法将尝试调用 registerWithMaster 方法（见代码清单 9-73）向 Master 注册 Application。如果执行 registerWithMaster 的过程中捕获到异常，那么会调用 markDisconnected 方法（见代码清单 9-69），将当前 ClientEndpoint 标记为和 Master 断开连接，并且调用 StandaloneAppClient 的 stop 方法（见代码清单 9-78）停止 StandaloneAppClient。

代码清单9-69　markDisconnected方法的实现

```
def markDisconnected() {
  if (!alreadyDisconnected) {
    listener.disconnected()
    alreadyDisconnected = true
  }
}
```

根据代码清单 9-69，markDisconnected 方法在判断 ClientEndpoint 的 alreadyDisconnected 状态不为 true 时，先调用外部类 StandaloneAppClient 的 listener 属性（即 StandaloneAppClientListener）的 disconnected 方法，然后将 alreadyDisconnected 设置为 true。

2. ClientEndpoint 的消息处理

ClientEndpoint 的 receive 方法能够接收 RegisteredApplication、ApplicationRemoved、ExecutorAdded、ExecutorUpdated、MasterChanged 等消息，如代码清单 9-70 所示。

代码清单9-70　ClientEndpoint的receive方法

```
override def receive: PartialFunction[Any, Unit] = {
  case RegisteredApplication(appId_, masterRef) =>
    appId.set(appId_)
    registered.set(true)
    master = Some(masterRef)
    listener.connected(appId.get)

  case ApplicationRemoved(message) =>
    markDead("Master removed our application: %s".format(message))
    stop()

  case ExecutorAdded(id: Int, workerId: String, hostPort: String, cores: Int,
      memory: Int) =>
    val fullId = appId + "/" + id
    logInfo("Executor added: %s on %s (%s) with %d cores".format(fullId, worker-
        Id, hostPort,
      cores))
    listener.executorAdded(fullId, workerId, hostPort, cores, memory)

  case ExecutorUpdated(id, state, message, exitStatus, workerLost) =>
    val fullId = appId + "/" + id
    val messageText = message.map(s => " (" + s + ")").getOrElse("")
    logInfo("Executor updated: %s is now %s%s".format(fullId, state, messageText))
    if (ExecutorState.isFinished(state)) {
      listener.executorRemoved(fullId, message.getOrElse(""), exitStatus, workerLost)
    }
```

```
    case MasterChanged(masterRef, masterWebUiUrl) =>
      logInfo("Master has changed, new master is at " + masterRef.address.toSparkURL)
      master = Some(masterRef)
      alreadyDisconnected = false
      masterRef.send(MasterChangeAcknowledged(appId.get))
  }
```

根据代码清单9-70可以看到，ClientEndpoint接收各种消息的处理都非常简单，这里就不赘述了。

除了receive方法，ClientEndpoint的receiveAndReply方法还能够处理StopAppClient、RequestExecutors、KillExecutors三种消息，如代码清单9-71所示。

代码清单9-71 ClientEndpoint的receiveAndReply方法

```
override def receiveAndReply(context: RpcCallContext): PartialFunction[Any, Unit] = {
  case StopAppClient =>
    markDead("Application has been stopped.")
    sendToMaster(UnregisterApplication(appId.get))
    context.reply(true)
    stop()

  case r: RequestExecutors =>
    master match {
      case Some(m) => askAndReplyAsync(m, context, r)
      case None =>
        logWarning("Attempted to request executors before registering with Master.")
        context.reply(false)
    }

  case k: KillExecutors =>
    master match {
      case Some(m) => askAndReplyAsync(m, context, k)
      case None =>
        logWarning("Attempted to kill executors before registering with Master.")
        context.reply(false)
    }
}
```

根据代码清单9-71，接收到StopAppClient消息后，ClientEndpoint最重要的一步是向Master发送UnregisterApplication消息。对于请求Executor（RequestExecutors）和“杀死”Executor（KillExecutors）两种消息，ClientEndpoint将通过调用askAndReplyAsync方法（见代码清单9-72）向Master转发这两种消息。

代码清单9-72 askAndReplyAsync方法的实现

```
private def askAndReplyAsync[T](
    endpointRef: RpcEndpointRef,
    context: RpcCallContext,
    msg: T): Unit = {
  endpointRef.ask[Boolean](msg).andThen {
    case Success(b) => context.reply(b)
    case Failure(ie: InterruptedException) => // Cancelled
```

```
    case Failure(NonFatal(t)) => context.sendFailure(t)
  }(ThreadUtils.sameThread)
}
```

3. 向 Master 注册应用信息

registerWithMaster 方法用于向 Master 注册 Application，其实现如代码清单 9-73 所示。

代码清单9-73 向Master注册应用信息

```
private def registerWithMaster(nthRetry: Int) {
  registerMasterFutures.set(tryRegisterAllMasters()) // 向所有的Master尝试注册
      Application
  registrationRetryTimer.set(registrationRetryThread.schedule(new Runnable {
    override def run(): Unit = {
      if (registered.get) { // 如果已经注册成功，那么取消向Master注册Application
        registerMasterFutures.get.foreach(_.cancel(true))
        registerMasterThreadPool.shutdownNow()
      } else if (nthRetry >= REGISTRATION_RETRIES) { // 重试次数超过限制，标记
          ClientEndpoint死亡
        markDead("All masters are unresponsive! Giving up.")
      } else {
        registerMasterFutures.get.foreach(_.cancel(true))
        registerWithMaster(nthRetry + 1) // 向Master注册Application的重试
      }
    }
  }, REGISTRATION_TIMEOUT_SECONDS, TimeUnit.SECONDS))
}
```

根据代码清单 9-73，registerWithMaster 方法的执行步骤如下。

1）调用 tryRegisterAllMasters 方法（见代码清单 9-74）向所有的 Master 尝试注册 Application，并将返回的 Future 保存到 registerMasterFutures。

2）向 registrationRetryThread 提交定时调度，调度间隔时长为 REGISTRATION_TIMEOUT_SECONDS。调度任务的执行逻辑是：如果已经注册成功，那么调用 registerMasterFutures 中保存的每一个 Future 的 cancel 方法取消向 Master 注册 Application；如果重试次数超过了 REGISTRATION_RETRIES 的限制，那么调用 markDead 方法（见代码清单 9-75）标记当前 ClientEndpoint 进入死亡状态；其他情况下首先调用 registerMasterFutures 中保存的每一个 Future 的 cancel 方法取消向 Master 注册 Application，然后再次调用 registerWithMaster（这次调用将把重试次数加 1）。

代码清单9-74 tryRegisterAllMasters方法的实现

```
private def tryRegisterAllMasters(): Array[JFuture[_]] = {
  for (masterAddress <- masterRpcAddresses) yield {
    registerMasterThreadPool.submit(new Runnable {
      override def run(): Unit = try {
        if (registered.get) {
          return
        }
        logInfo("Connecting to master " + masterAddress.toSparkURL + "...")
```

```
            val masterRef = rpcEnv.setupEndpointRef(masterAddress, Master.ENDPOINT_NAME)
            masterRef.send(RegisterApplication(appDescription, self))
          } catch {
            case ie: InterruptedException => // Cancelled
            case NonFatal(e) => logWarning(s"Failed to connect to master $master-
                Address", e)
          }
        })
      }
    }
```

根据代码清单9-74，tryRegisterAllMasters方法通过masterRpcAddresses中每个Master的RpcAddress，得到每个Master的RpcEndpointRef，然后通过这些RpcEndpointRef向每个Master发送RegisterApplication消息（此消息携带外部类StandaloneAppClient的appDescription属性和ClientEndpoint自身的RpcEndpointRef）。

代码清单9-75 markDead的实现

```
def markDead(reason: String) {
  if (!alreadyDead.get) {
    listener.dead(reason)
    alreadyDead.set(true)
  }
}
```

根据9.6.8节的内容我们知道，当注册Application成功后，Master将向ClientEndpoint回复RegisteredApplication消息，那么ClientEndpoint是如何处理RegisteredApplication消息的呢？代码清单9-70展示了ClientEndpoint对RegisteredApplication消息的处理实现，具体处理包括：将Master分配的Application ID设置到appId中，将registered设置为true，将Master的RpcEndpointRef更新到master、调用StandaloneAppClientListener（实际为StandaloneSchedulerBackend）的connected方法等。

StandaloneSchedulerBackend的connected方法的实现如代码清单9-76所示。

代码清单9-76 StandaloneSchedulerBackend的connected方法

```
override def connected(appId: String) {
  logInfo("Connected to Spark cluster with app ID " + appId)
  this.appId = appId
  notifyContext()
  launcherBackend.setAppId(appId)
}
```

根据代码清单9-76，StandaloneSchedulerBackend的connected方法首先将设置appId为Master分配的Application ID，然后调用notifyContext方法（见代码清单9-77）释放信号量，最后调用LauncherBackend的setAppId方法向LaunchServer发送appId。

代码清单9-77 notifyContext方法的实现

```
private def notifyContext() = {
```

```
    registrationBarrier.release()
  }
```

9.8.2 StandaloneAppClient 的实现分析

StandaloneAppClient 是 Application 与集群管理器进行对话的客户端。StandaloneAppClient 启动后才能正常工作，当不再需要与集群管理器通信时，需要停止它。StandaloneAppClient 最为核心的功能是向集群管理器请求或“杀死”Executor。下面将对这些内容一一进行介绍。

1. StandaloneAppClient 的启动

StandaloneAppClient 的 start 方法用于启动 StandaloneAppClient，代码如下。

```
def start() {
  endpoint.set(rpcEnv.setupEndpoint("AppClient", new ClientEndpoint(rpcEnv)))
}
```

根据上述代码，在启动 StandaloneAppClient 时只做了一件事——向 SparkContext 的 SparkEnv 的 RpcEnv 注册 ClientEndpoint，进而引起对 ClientEndpoint 的启动和向 Master 注册 Application。

2. StandaloneAppClient 的停止

StandaloneAppClient 的 stop 方法用于停止 StandaloneAppClient，如代码清单 9-78 所示。

代码清单9-78 StandaloneAppClient的stop方法

```
def stop() {
  if (endpoint.get != null) {
    try {
      val timeout = RpcUtils.askRpcTimeout(conf)
      timeout.awaitResult(endpoint.get.ask[Boolean](StopAppClient))
    } catch { // 忽略捕获到TimeoutException，打印日志的代码 }
    endpoint.set(null)
  }
}
```

根据 stop 方法的实现，我们知道停止 StandaloneAppClient 实际不过是向 ClientEndpoint 发送了 StopAppClient 消息，并将 ClientEndpoint 的 RpcEndpointRef 从 endpoint 中清除。

3. 请求 Executor 资源

StandaloneAppClient 的 requestTotalExecutors 方法（见代码清单 9-79）用于向 Master 请求所需的所有 Executor 资源。

代码清单9-79 requestTotalExecutors的实现

```
def requestTotalExecutors(requestedTotal: Int): Future[Boolean] = {
  if (endpoint.get != null && appId.get != null) {
    endpoint.get.ask[Boolean](RequestExecutors(appId.get, requestedTotal))
  } else {
```

```
      logWarning("Attempted to request executors before driver fully initialized.")
      Future.successful(false)
    }
  }
```

根据代码清单 9-79，requestTotalExecutors 方法只能在 Application 注册成功之后调用。requestTotalExecutors 方法将向 ClientEndpoint 发送 RequestExecutors 消息（此消息携带 Application ID 和所需的 Executor 总数）。根据对 ClientEndpoint 处理 RequestExecutors 消息的分析，我们知道 ClientEndpoint 将向 Master 转发 RequestExecutors 消息。Master 处理 RequestExecutors 消息的内容请回顾 9.6.9 节。

4. “杀死” Executor

StandaloneAppClient 的 killExecutors 方法（见代码清单 9-80）用于向 Master 请求“杀死” Executor。

代码清单9-80 killExecutors的实现

```
def killExecutors(executorIds: Seq[String]): Future[Boolean] = {
  if (endpoint.get != null && appId.get != null) {
    endpoint.get.ask[Boolean](KillExecutors(appId.get, executorIds))
  } else {
    logWarning("Attempted to kill executors before driver fully initialized.")
    Future.successful(false)
  }
}
```

根据代码清单 9-80，killExecutors 方法只能在 Application 注册成功之后调用。killExecutors 方法将向 ClientEndpoint 发送 KillExecutors 消息（此消息携带 Application ID 和要“杀死”的 Executor 的 ID）。

9.9 StandaloneSchedulerBackend 的实现分析

在 7.8.2 节我们曾经介绍了 local 部署模式下，SchedulerBackend 的实现类 LocalSchedulerBackend，本节将介绍在 local-cluster 模式和 Standalone 模式下，SchedulerBackend 的另一个实现类 StandaloneSchedulerBackend。由于 StandaloneSchedulerBackend 继承自 CoarseGrainedSchedulerBackend，本节还需要介绍 CoarseGrainedSchedulerBackend 及其与其他组件通信的内部类 DriverEndpoint。

9.9.1 StandaloneSchedulerBackend 的属性

为便于分析，本节首先来熟悉 StandaloneSchedulerBackend 包含的属性。StandaloneSchedulerBackend 的属性包括从父类 CoarseGrainedSchedulerBackend 继承的属性，也包括自身的属性。从 CoarseGrainedSchedulerBackend 继承的属性如下。

- scheduler：即 TaskSchedulerImpl。
- rpcEnv：即 RpcEnv。
- totalCoreCount：用于统计分配给 Driver 的内核总数。
- totalRegisteredExecutors：当前注册到 CoarseGrainedSchedulerBackend 的 Executor 的总数。
- conf：即 SparkConf。
- maxRpcMessageSize：RPC 消息的最大大小。可通过 spark.rpc.message.maxSize 属性配置，默认为 128MB。
- defaultAskTimeout：类型为 RpcTimeout，表示 RPC 请求的超时时间。可通过 spark.rpc.askTimeout 属性或 spark.network.timeout 属性配置，默认为 120s。
- _minRegisteredRatio：已经注册的资源与期望得到的资源之间的最小比值，当比值大于等于 _minRegisteredRatio 时，才提交 Task。可通过 spark.scheduler.minRegisteredResourcesRatio 属性配置，默认为 0。
- maxRegisteredWaitingTimeMs：在还未达到 _minRegisteredRatio 时，如果已经等待了超过 maxRegisteredWaitingTimeMs 指定的时间，那么提交 Task。可通过 spark.scheduler.maxRegisteredResourcesWaitingTime 属性配置，默认为 30s。
- createTime：CoarseGrainedSchedulerBackend 的创建时间。
- executorDataMap：Executor 的 ID 与 ExecutorData 之间的映射关系缓存。ExecutorData 保存了 Executor 的 RpcEndpointRef、RpcAddress、Host、Executor 的空闲内核数（freeCores）、Executor 的总内核数（totalCores）等信息。
- numPendingExecutors：从集群管理器请求的还未注册到 CoarseGrainedSchedulerBackend 的 Executor 的数量。
- listenerBus：即 LiveListenerBus。
- executorsPendingToRemove：请求集群管理器 kill 一个 Executor 时，Executor 并不会立即被“杀死”，所以 executorsPendingToRemove 缓存那些请求 kill 的 Executor 的 ID 与是否被“杀死”之间的映射关系。
- hostToLocalTaskCount：缓存机器的 Host 和在机器本地运行的 Task 数量之间的映射关系。
- localityAwareTasks：用于统计有本地性需求的 Task 的数量。
- currentExecutorIdCounter：用于保存注册到 executorDataMap 的最大的 Executor 的 ID。

再来看看 StandaloneSchedulerBackend 自身的属性。

- client：即 StandaloneAppClient。
- stopping：标记 StandaloneSchedulerBackend 是否正在停止。
- launcherBackend：即运行器后端接口 LauncherBackend。具体实现与 LocalSchedulerBackend 的 launcherBackend 属性的实现类似。

- appId：Application 的 ID。
- registrationBarrier：使用 Java 的信号量（Semaphore）实现的栅栏，用于等待 Application 向 Master 注册完成后，将 Application 的当前状态（此时为正在运行，即 RUNNING）告知 LauncherServer。
- maxCores：Application 可以申请获得的最大内核数。可通过 spark.cores.max 属性配置。
- totalExpectedCores：Application 期望获得的内核数，如果设置了 maxCores，则为 maxCores，否则为 0。

9.9.2 DriverEndpoint 的实现分析

DriverEndpoint 是 CoarseGrainedSchedulerBackend 的内部类，无论 CoarseGrainedSchedulerBackend 还是 StandaloneSchedulerBackend，都借助于 DriverEndpoint 和其他组件通信，所以我们先来分析它。

DriverEndpoint 包含以下属性。

- rpcEnv：即 SparkContext 的 SparkEnv 中的 RpcEnv。
- sparkProperties：从 SparkConf 中获取的所有以 spark. 开头的属性信息。
- executorsPendingLossReason：已经丢失的（但是还不知道真实的退出原因）的 Executor 的 ID。
- ser：SparkEnv 的子组件 closureSerializer 的实例，用于对闭包序列化。
- addressToExecutorId：每个 Executor 的 RpcEnv 的地址与 Executor 的 ID 之间的映射关系。
- reviveThread：只有一个线程的 ScheduledThreadPoolExecutor，用于唤起对延迟调度的 Task 进行资源分配与调度。

DriverEndpoint 中有很多功能实现，笔者挑选其中最为重要的启动、Task 状态更新、注册 Executor 进行分析，其他的功能（如停止 Driver、停止 Executor、移除 Executor 等）留给感兴趣的读者自行研究，笔者相信有了对启动、Task 状态更新、注册 Executor 的了解，其他功能的分析都将水到渠成。

1. DriverEndpoint 的启动

有了对 DriverEndpoint 中属性的了解，现在来看看 DriverEndpoint 的启动过程。由于 DriverEndpoint 继承自 ThreadSafeRpcEndpoint，所以将 DriverEndpoint 注册到 RpcEnv 的 Dispatcher 时，会触发对 DriverEndpoint 的 onStart 方法的调用。DriverEndpoint 的 onStart 方法的实现如代码清单 9-81 所示。

代码清单9-81 DriverEndpoint的onStart方法

```
override def onStart() {
```

仅供非商业用途或交流学习使用


```
    val reviveIntervalMs = conf.getTimeAsMs("spark.scheduler.revive.interval", "1s")
    reviveThread.scheduleAtFixedRate(new Runnable {
      override def run(): Unit = Utils.tryLogNonFatalError {
        Option(self).foreach(_.send(ReviveOffers))
      }
    }, 0, reviveIntervalMs, TimeUnit.MILLISECONDS)
  }
```

根据代码清单 9-81，DriverEndpoint 的 onStart 方法中主要向 reviveThread 提交了一个向 DriverEndpoint 自己发送 ReviveOffers 消息的定时任务。此定时任务的执行间隔可通过 spark.scheduler.revive.interval 属性配置，默认为 1s。DriverEndpoint 接收到 ReviveOffers 消息后的处理如下。

```
case ReviveOffers =>
  makeOffers()
```

DriverEndpoint 的 makeOffers 方法的实现如代码清单 9-82 所示。

代码清单9-82　DriverEndpoint的makeOffers方法

```
private def makeOffers() {
  val activeExecutors = executorDataMap.filterKeys(executorIsAlive)
  val workOffers = activeExecutors.map { case (id, executorData) =>
    new WorkerOffer(id, executorData.executorHost, executorData.freeCores)
  }.toIndexedSeq
  launchTasks(scheduler.resourceOffers(workOffers))
}
```



根据代码清单 9-82，DriverEndpoint 的 makeOffers 方法的执行步骤如下。

1）过滤出激活的 Executor。

2）根据每个激活的 Executor 的配置，创建 WorkerOffer。

3）调用 TaskSchedulerImpl 的 resourceOffers 方法（见代码清单 7-101）给 Task 分配资源。

4）调用 launchTasks 方法（见代码清单 9-83）运行 Task。

代码清单9-83　DriverEndpoint的launchTasks方法

```
private def launchTasks(tasks: Seq[Seq[TaskDescription]]) {
  for (task <- tasks.flatten) {
    val serializedTask = ser.serialize(task) // 对TaskDescription进行序列化
    if (serializedTask.limit >= maxRpcMessageSize) { // 序列化后的大小超出了Rpc消息的限制
      scheduler.taskIdToTaskSetManager.get(task.taskId).foreach { taskSetMgr =>
        try {
          var msg = "Serialized task %s:%d was %d bytes, which exceeds max allowed: " +
            "spark.rpc.message.maxSize (%d bytes). Consider increasing " +
            "spark.rpc.message.maxSize or using broadcast variables for large values."
          msg = msg.format(task.taskId, task.index, serializedTask.limit, maxRpc-
             MessageSize)
          taskSetMgr.abort(msg) // 放弃对TaskSetManager的调度
        } catch {
          case e: Exception => logError("Exception in error callback", e)
        }
```

仅供非商业用途或交流学习使用


```
        }
      }
      else {
        val executorData = executorDataMap(task.executorId)
        executorData.freeCores -= scheduler.CPUS_PER_TASK
        executorData.executorEndpoint.send(LaunchTask(new SerializableBuffer(seria
            lizedTask)))
      }
    }
  }
```

根据代码清单 9-83，launchTasks 方法遍历每个 TaskDescription，并对每个 TaskDescription 执行以下操作。

1）对 TaskDescription 进行序列化。

2）如果序列化后的 TaskDescription 的大小大于等于 RPC 消息大小的最大值 maxRpcMessageSize，则执行如下操作。

① 从 TaskSchedulerImpl 的 taskIdToTaskSetManager 中找出 Task 对应的 TaskSetManager。

② 调用 TaskSetManager 的 abort 方法放弃对 TaskSetManager 的调度。

3）如果序列化后的 TaskDescription 的大小小于 RPC 消息大小的最大值 maxRpcMessageSize，则执行如下操作。

① 减少 Executor 的空闲内核数 freeCores。

② 向 CoarseGrainedExecutorBackend 发送 LaunchTask 消息。CoarseGrainedExecutorBackend 将在收到 LaunchTask 消息后运行 Task。

2. DriverEndpoint 与 Task 状态更新

Task 在运行的过程中，会向 DriverEndpoint 发送 StatusUpdate 消息，这可以让 Driver 知道 Task 的当前状态，从而执行更新度量、将 Task 释放的资源分配给其他 Task 等操作。DriverEndpoint 的 receive 方法中实现了对 StatusUpdate 消息的接收处理，如代码清单 9-84 所示。

代码清单9-84 DriverEndpoint对StatusUpdate消息的处理

```
case StatusUpdate(executorId, taskId, state, data) =>
  scheduler.statusUpdate(taskId, state, data.value)
  if (TaskState.isFinished(state)) {
    executorDataMap.get(executorId) match {
      case Some(executorInfo) =>
        executorInfo.freeCores += scheduler.CPUS_PER_TASK
        makeOffers(executorId) // 给下一个要调度的Task分配资源并运行Task
      case None => // 对于未知的Executor，DriverEndpoint选择忽略
    }
  }
```

根据代码清单 9-84，DriverEndpoint 对 StatusUpdate 消息的处理步骤如下。

1）调用 TaskSchedulerImpl 的 statusUpdate 方法（见代码清单 7-103）更新 Task 的状态。

仅供非商业用途或交流学习使用

2）如果 Task 的状态是已经完成，则将 Task 释放的内核数增加到对应 Executor 的空闲内核数（freeCores），然后调用 makeOffers 方法给下一个要调度的 Task 分配资源并运行 Task。

3. 注册 Executor

给 Driver 分配了 Executor 之后，需要向 Driver 注册 Executor，以便于 Driver 与 Executor 之间直接通信，而不用通过 Worker 作为媒介，进而提升执行效率。DriverEndpoint 的 receiveAndReply 方法中实现了对 RegisterExecutor 消息的处理，代码如下。

```
case RegisterExecutor(executorId, executorRef, hostname, cores, logUrls) =>
  if (executorDataMap.contains(executorId)) { // 重复注册Executor
    executorRef.send(RegisterExecutorFailed("Duplicate executor ID: " + executorId))
    context.reply(true)
  } else {
    val executorAddress = if (executorRef.address != null) {
        executorRef.address
      } else {
        context.senderAddress
      }
    logInfo(s"Registered executor $executorRef ($executorAddress) with ID $executor-
        Id")
    addressToExecutorId(executorAddress) = executorId
    totalCoreCount.addAndGet(cores) // 增加Driver已经获得的内核总数
    totalRegisteredExecutors.addAndGet(1) // 向Driver注册的所有Executor的总数
    val data = new ExecutorData(executorRef, executorRef.address, hostname,
      cores, cores, logUrls)
    CoarseGrainedSchedulerBackend.this.synchronized {
      executorDataMap.put(executorId, data)
      if (currentExecutorIdCounter < executorId.toInt) {
        currentExecutorIdCounter = executorId.toInt
      }
      if (numPendingExecutors > 0) {
        numPendingExecutors -= 1
        logDebug(s"Decremented number of pending executors ($numPendingExecutors
            left)")
      }
    }
    executorRef.send(RegisteredExecutor)
    context.reply(true)
    listenerBus.post(
      SparkListenerExecutorAdded(System.currentTimeMillis(), executorId, data))
    makeOffers() // 给Task分配资源并运行Task
  }
```

根据上述代码，DriverEndpoint 处理 RegisterExecutor 消息的步骤如下。

1）如果外部类 CoarseGrainedSchedulerBackend 的 executorDataMap 中已经注册了 RegisterExecutor 消息携带的 executorId，那么通过向 CoarseGrainedExecutorBackend 回复 RegisterExecutorFailed 消息，告诉后者重复注册了。

2）如果外部类 CoarseGrainedSchedulerBackend 的 executorDataMap 中还未注册 RegisterExecutor 消息携带的 executorId，则执行如下操作。

① 获取 CoarseGrainedExecutorBackend 的 RpcAddress。

仅供非商业用途或交流学习使用

② 将 CoarseGrainedExecutorBackend 的 RpcAddress 和 executorId 的对应关系放入外部类 CoarseGrainedSchedulerBackend 的 addressToExecutorId 缓存中。

③ 增加外部类 CoarseGrainedSchedulerBackend 的 totalCoreCount，以表示 Driver 已经获得的内核总数。增加外部类 CoarseGrainedSchedulerBackend 的 totalRegisteredExecutors，以表示向 Driver 注册的所有 Executor 的总数。

④ 创建 ExecutorData 对象，并将 executorId 与 ExecutorData 的对应关系放入外部类 CoarseGrainedSchedulerBackend 的 executorDataMap 缓存中。此外，还将更新外部类 CoarseGrainedSchedulerBackend 的 currentExecutorIdCounter 和 numPendingExecutors 等属性。

⑤ 向 CoarseGrainedExecutorBackend 发送 RegisteredExecutor 消息。CoarseGrainedExecutorBackend 接收到 RegisteredExecutor 消息后将创建 Executor 实例（见代码清单 9-91）。

⑥ 向 LiveListenerBus 投递 SparkListenerExecutorAdded 事件。

⑦ 最后调用 makeOffers 方法（见代码清单 9-82）给 Task 分配资源并运行 Task。

4. 对 RetrieveSparkAppConfig 消息的处理

CoarseGrainedExecutorBackend 进程在启动过程中会向 DriverEndpoint 发送 RetrieveSparkAppConfig 消息，从 Driver 获取 Executor 所需的 Spark 属性信息和密钥信息。DriverEndpoint 的 receiveAndReply 方法中实现了对 RetrieveSparkAppConfig 消息的处理，代码如下。



```
case RetrieveSparkAppConfig =>
  val reply = SparkAppConfig(sparkProperties,
    SparkEnv.get.securityManager.getIOEncryptionKey())
  context.reply(reply)
```

5. 停止 Driver

DriverEndpoint 的 receiveAndReply 方法中实现了对 StopDriver 消息的处理，代码如下。

```
case StopDriver =>
  context.reply(true)
  stop()
```

可以看到，DriverEndpoint 向发送端回复 true 之后调用父类 RpcEndpoint 的 stop 方法（见代码清单 5-2）。

6. 停止 Executor

DriverEndpoint 的 receiveAndReply 方法中实现了对 StopExecutors 消息的处理，代码如下。

```
case StopExecutors =>
  logInfo("Asking each executor to shut down")
  for ((_, executorData) <- executorDataMap) {
    executorData.executorEndpoint.send(StopExecutor)
  }
  context.reply(true)
```

仅供非商业用途或交流学习使用

上面的代码遍历 executorDataMap 中的 ExecutorData，向每个 CoarseGrained-ExecutorBackend 进程发送 StopExecutor 消息。CoarseGrainedExecutorBackend 进程收到 StopExecutor 消息后的资源回收处理如代码清单 9-91 所示。

9.9.3 StandaloneSchedulerBackend 的启动

由于 StandaloneSchedulerBackend 的 start 方法会率先调用父类 CoarseGrainedSchedulerBackend 的 start 方法，所以我们先来看看 CoarseGrainedSchedulerBackend 的 start 方法的实现，如代码清单 9-85 所示。

代码清单9-85 CoarseGrainedSchedulerBackend的start方法

```
override def start() {
  val properties = new ArrayBuffer[(String, String)]
  for ((key, value) <- scheduler.sc.conf.getAll) {
    if (key.startsWith("spark.")) {
      properties += ((key, value)) // 将以spark.开头的属性添加到数组缓冲properties中
    }
  }

  driverEndpoint = createDriverEndpointRef(properties) // 创建并注册DriverEndpoint
}

protected def createDriverEndpointRef(
    properties: ArrayBuffer[(String, String)]): RpcEndpointRef = {
  rpcEnv.setupEndpoint(ENDPOINT_NAME, createDriverEndpoint(properties))
}

protected def createDriverEndpoint(properties: Seq[(String, String)]): DriverEndpoint
    = {
  new DriverEndpoint(rpcEnv, properties)
}
```

根据代码清单 9-85，CoarseGrainedSchedulerBackend 的 start 方法的执行步骤如下。

1）从 SparkContext 的 SparkConf 中将以 spark. 开头的属性添加到数组缓冲 properties 中。

2）调用 createDriverEndpointRef 方法创建 DriverEndpoint，并将 DriverEndpoint 注册到 SparkContext 的 SparkEnv 的 RpcEnv 中，注册时以常量 ENDPOINT_NAME（值为 CoarseGrainedScheduler）作为注册名。

了解了父类的 start 方法，现在来看看 StandaloneSchedulerBackend 的 start 方法的实现，如代码清单 9-86 所示。

代码清单9-86 StandaloneSchedulerBackend的start方法

```
override def start() {
  super.start() // 创建并注册DriverEndpoint
  launcherBackend.connect() // 与LauncherServer建立连接

  val driverUrl = RpcEndpointAddress( // 生成Driver URL，Executor将通过此URL与Driver
                                      通信
```

仅供非商业用途或交流学习使用


```
      sc.conf.get("spark.driver.host"),
      sc.conf.get("spark.driver.port").toInt,
      CoarseGrainedSchedulerBackend.ENDPOINT_NAME).toString
    val args = Seq(
      "--driver-url", driverUrl,
      "--executor-id", "{{EXECUTOR_ID}}",
      "--hostname", "{{HOSTNAME}}",
      "--cores", "{{CORES}}",
      "--app-id", "{{APP_ID}}",
      "--worker-url", "{{WORKER_URL}}")
    val extraJavaOpts = sc.conf.getOption("spark.executor.extraJavaOptions")
      .map(Utils.splitCommandString).getOrElse(Seq.empty)
    val classPathEntries = sc.conf.getOption("spark.executor.extraClassPath")
      .map(_.split(java.io.File.pathSeparator).toSeq).getOrElse(Nil)
    val libraryPathEntries = sc.conf.getOption("spark.executor.extraLibraryPath")
      .map(_.split(java.io.File.pathSeparator).toSeq).getOrElse(Nil)

    val testingClassPath =
      if (sys.props.contains("spark.testing")) {
        sys.props("java.class.path").split(java.io.File.pathSeparator).toSeq
      } else {
        Nil
      }

    val sparkJavaOpts = Utils.sparkJavaOpts(conf, SparkConf.isExecutorStartupConf)
    val javaOpts = sparkJavaOpts ++ extraJavaOpts
    val command = Command("org.apache.spark.executor.CoarseGrainedExecutorBackend",
  //创建Command
      args, sc.executorEnvs, classPathEntries ++ testingClassPath, libraryPathEntries,
          javaOpts)
    val appUIAddress = sc.ui.map(_.appUIAddress).getOrElse("")
    val coresPerExecutor = conf.getOption("spark.executor.cores").map(_.toInt)
    val initialExecutorLimit =
      if (Utils.isDynamicAllocationEnabled(conf)) {
        Some(0)
      } else {
        None
      }
    val appDesc = new ApplicationDescription(sc.appName, maxCores, sc.executorMemory,
       command,
      appUIAddress, sc.eventLogDir, sc.eventLogCodec, coresPerExecutor, initial-
          ExecutorLimit)
    client = new StandaloneAppClient(sc.env.rpcEnv, masters, appDesc, this, conf)
    client.start() // 启动StandaloneAppClient
    launcherBackend.setState(SparkAppHandle.State.SUBMITTED)
    waitForRegistration() // 等待注册Application完成
    launcherBackend.setState(SparkAppHandle.State.RUNNING)
  }
```

根据代码清单 9-86，StandaloneSchedulerBackend 的 start 方法的执行步骤如下。

1）调用父类 CoarseGrainedSchedulerBackend 的 start 方法（见代码清单 9-85）创建并注册 DriverEndpoint。

2）调用 LauncherBackend 的 connect 方法（见代码清单 7-83）与 LauncherServer 建立

仅供非商业用途或交流学习使用

连接。

3）生成 Driver URL，格式为 spark://CoarseGrainedScheduler@ ${driverHost}:${driverPort}。Executor 将通过此 URL 与 Driver 通信。

4）拼接参数列表 args、获取额外的 Java 参数 extraJavaOpts（通过 spark.executor.extraJavaOptions 属性配置）、额外的类路径 classPathEntries（通过 spark.executor.extraClassPath 属性配置）、额外的库路径 libraryPathEntries（通过 spark.executor.extraLibraryPath 属性配置）等。

5）从 SparkConf 中获取需要传递给 Executor 用于启动的配置 sparkJavaOpts。这些配置包括：以 spark.auth 开头的配置（但不包括 spark.authenticate.secret）、以 spark.ssl 开头的配置、以 spark.rpc 开头的配置、以 spark. 开头且以 .port 结尾的配置及以 spark.port. 开头的配置。

6）将 extraJavaOpts 与 sparkJavaOpts 合并到 javaOpts。

7）创建 Command 对象。样例类 Command 定义了执行 Executor 的命令。这里以 org.apache.spark.executor.CoarseGrainedExecutorBackend 作为 Command 的 mainClass 属性、以 args 作为 Command 的 arguments 属性、以 SparkContext 的 executorEnvs 属性作为 Command 的 environment 属性、以 classPathEntries 作为 Command 的 classPathEntries 属性、以 libraryPathEntries 作为 Command 的 libraryPathEntries 属性、以 javaOpts 作为 Command 的 javaOpts 属性。



8）获取 Spark UI 的 http 地址 appUIAddress、获取每个 Executor 分配的内核数 coresPerExecutor（可通过 spark.executor.cores 属性配置）、Executor 的初始限制 initialExecutorLimit（如果启用了动态分配 Executor，那么 initialExecutorLimit 被设置为 0，ExecutorAllocationManager 之后会将真实的初始限制值传递给 Master）。

9）创建 Application 描述信息（ApplicationDescription）。

10）创建并启动 StandaloneAppClient。根据 9.8 节的内容我们知道，StandaloneAppClient 的 start 方法将创建 ClientEndpoint，并向 SparkContext 的 SparkEnv 的 RpcEnv 注册 ClientEndpoint，进而引起对 ClientEndpoint 的启动和向 Master 注册 Application。

11）调用 LauncherBackend 的 setState 方法（见代码清单 7-85）向 LauncherServer 传递应用已经提交（SUBMITTED）的状态。

12）调用 waitForRegistration 方法（见代码清单 9-87）等待注册 Application 完成。根据 9.6.8 节的内容，我们知道当注册 Application 成功后，Master 将向 ClientEndpoint 发送 RegisteredApplication 消息，进而调用 StandaloneSchedulerBackend 的 connected 方法释放信号量，这样 waitForRegistration 方法将可以获得信号量。

13）调用 LauncherBackend 的 setState 方法（见代码清单 7-85）向 LauncherServer 传递应用正在运行（RUNNING）的状态。

仅供非商业用途或交流学习使用

代码清单9-87 waitForRegistration方法的实现

```
private def waitForRegistration() = {
  registrationBarrier.acquire()
}
```

9.9.4 StandaloneSchedulerBackend 的停止

StandaloneSchedulerBackend 的 stop 方法用于停止 StandaloneSchedulerBackend，其实现如下。

```
override def stop(): Unit = synchronized {
  stop(SparkAppHandle.State.FINISHED)
}
```

上面的 stop 方法调用了重载的 stop 方法，重载的 stop 方法如代码清单 9-88 所示。

代码清单9-88 StandaloneSchedulerBackend中重载的stop方法

```
private def stop(finalState: SparkAppHandle.State): Unit = synchronized {
  try {
    stopping = true

    super.stop()
    client.stop()

    val callback = shutdownCallback
    if (callback != null) {
      callback(this)
    }
  } finally {
    launcherBackend.setState(finalState)
    launcherBackend.close()
  }
}
```



根据代码清单 9-88，stop 方法首先调用 StandaloneAppClient 的 stop 方法（见代码清单 9-78）停止 StandaloneAppClient，然后调用父类 CoarseGrainedSchedulerBackend 的 stop 方法，最后回调关闭函数。CoarseGrainedSchedulerBackend 的 stop 方法的实现如代码清单 9-89 所示。

代码清单9-89 CoarseGrainedSchedulerBackend的stop方法

```
def stopExecutors() {
  try {
    if (driverEndpoint != null) {
      logInfo("Shutting down all executors")
      driverEndpoint.askWithRetry[Boolean](StopExecutors)
    }
  } catch {
    // 忽略对异常的处理
  }
}
```

仅供非商业用途或交流学习使用


```
override def stop() {
  stopExecutors()
  try {
    if (driverEndpoint != null) {
      driverEndpoint.askWithRetry[Boolean](StopDriver)
    }
  } catch {
    // 忽略对异常的处理
  }
}
```

根据代码清单 9-89，CoarseGrainedSchedulerBackend 的 stop 方法首先调用 stopExecutors 方法（实际向 DriverEndpoint 发送 StopExecutors 消息）停止 Executor，然后向 DriverEndpoint 发送 StopDriver 消息停止 Driver。

9.9.5 StandaloneSchedulerBackend 与资源分配

SchedulerBackend 存在的最大价值是代理 TaskSchedulerImpl，将集群分配给 Application 的资源进一步分配给 Task。本节将介绍在 Standalone 模式下进行资源的二级分配的组件，即 SchedulerBackend 的实现类 StandaloneSchedulerBackend。

1. StandaloneSchedulerBackend 的二级资源分配

根据 7.10.4 节对 TaskSchedulerImpl 的提交 Task 的实现分析，我们知道 TaskSchedulerImpl 的 submitTasks 方法最后会调用 SchedulerBackend 的 reviveOffers 方法给 Task 分配资源并运行 Task。StandaloneSchedulerBackend 的父类 CoarseGrainedSchedulerBackend 实现了这一方法，代码如下。

```
override def reviveOffers() {
  driverEndpoint.send(ReviveOffers)
}
```

根据上述代码，CoarseGrainedSchedulerBackend 的实现非常简单——只是向 DriverEndpoint 发送了 ReviveOffers 消息。根据之前对 DriverEndpoint 的启动的分析，我们知道 DriverEndpoint 收到 ReviveOffers 消息后，将调用 makeOffers 方法给 Task 分配资源并运行 Task。

2. StandaloneSchedulerBackend 与 ExecutorAllocationManager

根据 4.9 节的内容，只有在 SchedulerBackend 的实现类同时实现了特质 ExecutorAllocationClient 的情况下，才会创建 ExecutorAllocationManager。SchedulerBackend 的实现类中只有 CoarseGrainedSchedulerBackend 同时实现了特质 ExecutorAllocationClient。在启动 ExecutorAllocationManager 的最后，会调用 CoarseGrainedSchedulerBackend 的 requestTotalExecutors 方法，其实现如下。

```
final override def requestTotalExecutors(
    numExecutors: Int, localityAwareTasks: Int, hostToLocalTaskCount: Map[String,
```

```
        Int]
    ): Boolean = {
    if (numExecutors < 0) { // 忽略抛出IllegalArgumentException的代码}

    val response = synchronized {
      this.localityAwareTasks = localityAwareTasks
      this.hostToLocalTaskCount = hostToLocalTaskCount

      numPendingExecutors =
        math.max(numExecutors - numExistingExecutors + executorsPendingToRemove.
            size, 0)

      doRequestTotalExecutors(numExecutors)
    }

    defaultAskTimeout.awaitResult(response)
  }
```

requestTotalExecutors方法一共有三个参数，分别是请求的Executor数量（numExecutors）、有本地性偏好的Task总数（localityAwareTasks）、Host与想要在Host本地运行的Task数量之间的映射关系（hostToLocalTaskCount）。requestTotalExecutors方法计算了待申请的Executor数量（numPendingExecutors）后调用doRequestTotalExecutors方法。StandaloneSchedulerBackend实现了父类CoarseGrainedSchedulerBackend的doRequestTotalExecutors方法，代码如下。

```
protected override def doRequestTotalExecutors(requestedTotal: Int): Future[Boolean]
    = {
  Option(client) match {
    case Some(c) => c.requestTotalExecutors(requestedTotal)
    case None =>
      logWarning("Attempted to request executors before driver fully initialized.")
      Future.successful(false)
  }
}
```

上面的doRequestTotalExecutors方法主要调用了StandaloneAppClient的requestTotalExecutors方法（见代码清单9-79）向Master请求所需的所有Executor资源。

9.10 CoarseGrainedExecutorBackend详解

在9.7.6节曾经提到过CoarseGrainedExecutorBackend，而且还给出了CoarseGrainedExecutorBackend进程的截图，本节将对CoarseGrainedExecutorBackend的实现进行详细分析。CoarseGrainedExecutorBackend是Driver与Executor通信的后端接口，但只在local-cluster和Standalone模式下才会使用。由于CoarseGrainedExecutorBackend作为单独的进程存在，所以CoarseGrainedExecutorBackend的伴生对象中实现了main方法。在main方法中将创建CoarseGrainedExecutorBackend实例，因此本节还需要介绍CoarseGrained-

ExecutorBackend。对于 CoarseGrainedExecutorBackend，我们首先介绍它的属性，然后挑选其最核心的功能进行分析。

9.10.1 CoarseGrainedExecutorBackend 进程

CoarseGrainedExecutorBackend 的伴生对象中实现了 main 方法，所以可以作为单独的 Java 进程启动。main 方法的实现如下。

```
def main(args: Array[String]) {
  var driverUrl: String = null
  var executorId: String = null
  var hostname: String = null
  var cores: Int = 0
  var appId: String = null
  var workerUrl: Option[String] = None
  val userClassPath = new mutable.ListBuffer[URL]()
  var argv = args.toList
  while (!argv.isEmpty) {
    argv match {
      case ("--driver-url") :: value :: tail =>
        driverUrl = value
        argv = tail
      case ("--executor-id") :: value :: tail =>
        executorId = value
        argv = tail
      case ("--hostname") :: value :: tail =>
        hostname = value
        argv = tail
      case ("--cores") :: value :: tail =>
        cores = value.toInt
        argv = tail
      case ("--app-id") :: value :: tail =>
        appId = value
        argv = tail
      case ("--worker-url") :: value :: tail =>
        workerUrl = Some(value)
        argv = tail
      case ("--user-class-path") :: value :: tail =>
        userClassPath += new URL(value)
        argv = tail
      case Nil =>
      case tail =>
        System.err.println(s"Unrecognized options: ${tail.mkString(" ")}")
        printUsageAndExit()
    }
  }

  if (driverUrl == null || executorId == null || hostname == null || cores <= 0 ||
    appId == null) {
    printUsageAndExit()
  }

  run(driverUrl, executorId, hostname, cores, appId, workerUrl, userClassPath)
```

```
  System.exit(0)
}
```

根据 main 方法的代码实现，其执行步骤如下。

1）对执行参数进行解析，如将参数 --driver-url 的值解析后保存到变量 driverUrl 中。

2）将解析得到的变量作为参数，调用 run 方法。run 方法的实现如代码清单 9-90 所示。

代码清单9-90 CoarseGrainedExecutorBackend伴生对象的run方法

```
private def run(
    driverUrl: String,
    executorId: String,
    hostname: String,
    cores: Int,
    appId: String,
    workerUrl: Option[String],
    userClassPath: Seq[URL]) {

  Utils.initDaemon(log)

  SparkHadoopUtil.get.runAsSparkUser { () =>
    Utils.checkHost(hostname)
    // 从CoarseGrainedSchedulerBackend获取所需的Spark属性信息和密钥
    val executorConf = new SparkConf
    val port = executorConf.getInt("spark.executor.port", 0)
    val fetcher = RpcEnv.create("driverPropsFetcher", hostname, port, executor-
      Conf,
      new SecurityManager(executorConf), clientMode = true)
    val driver = fetcher.setupEndpointRefByURI(driverUrl)
    val cfg = driver.askWithRetry[SparkAppConfig](RetrieveSparkAppConfig)
    val props = cfg.sparkProperties ++ Seq[(String, String)](("spark.app.id",
      appId))
    fetcher.shutdown()

    val driverConf = new SparkConf() // 创建Executor自己的SparkConf
    for ((key, value) <- props) {
      if (SparkConf.isExecutorStartupConf(key)) {
        driverConf.setIfMissing(key, value)
      } else {
        driverConf.set(key, value)
      }
    }
    if (driverConf.contains("spark.yarn.credentials.file")) {
      logInfo("Will periodically update credentials from: " +
        driverConf.get("spark.yarn.credentials.file"))
      SparkHadoopUtil.get.startCredentialUpdater(driverConf)
    }

    val env = SparkEnv.createExecutorEnv( // 创建Executor自身的SparkEnv
      driverConf, executorId, hostname, port, cores, cfg.ioEncryptionKey, isLocal
        = false)
    // 创建并注册CoarseGrainedExecutorBackend实例
    env.rpcEnv.setupEndpoint("Executor", new CoarseGrainedExecutorBackend(
      env.rpcEnv, driverUrl, executorId, hostname, cores, userClassPath, env))
    workerUrl.foreach { url => // 创建并注册WorkerWatcher实例
```

```
        env.rpcEnv.setupEndpoint("WorkerWatcher", new WorkerWatcher(env.rpcEnv, url))
      }
      env.rpcEnv.awaitTermination()
      SparkHadoopUtil.get.stopCredentialUpdater()
    }
  }
```

根据代码清单 9-90，run 方法的执行步骤如下。

1）创建名为 driverPropsFetcher 的 RpcEnv。此 RpcEnv 主要用于从 Driver 拉取属性信息，并非 Executor 的 SparkEnv 中的 RpcEnv。

2）向 DriverEndpoint 发送 RetrieveSparkAppConfig 消息，从 CoarseGrainedSchedulerBackend 获取所需的 Spark 属性信息和密钥。DriverEndpoint 对 RetrieveSparkAppConfig 消息的处理已在 9.9.2 节介绍。

3）将 CoarseGrainedSchedulerBackend 回复的 SparkAppConfig 消息中携带的 Spark 属性和 Application 的 ID 属性保存到 props 中。

4）由于名为 driverPropsFetcher 的 RpcEnv 已经完成了它的使命，因此关闭它。

5）创建 Executor 自己的 SparkConf。变量名 driverConf 说明 SparkConf 都是从 Driver 获取的。

6）将 props 中的属性全部保存到 SparkConf 中。

7）调用 SparkEnv 伴生对象的 createExecutorEnv 方法创建 Executor 自身的 SparkEnv。createExecutorEnv 方法与 4.2 节介绍的 SparkEnv 伴生对象的 createDriverEnv 方法一样，实际也是调用了 SparkEnv 伴生对象的 create 方法。

8）创建 CoarseGrainedExecutorBackend 实例，并注册到 Executor 自身 SparkEnv 的子组件 RpcEnv 中。

9）创建 WorkerWatcher 实例，并注册到 Executor 自身 SparkEnv 的子组件 RpcEnv 中。顾名思义，WorkerWatcher 是 Worker 的守望者，当发生 Worker 进程退出、连接断开、网络出错等情况时，终止 CoarseGrainedExecutorBackend 进程。Worker 与 CoarseGrainedExecutorBackend 进程相辅相成。由于 WorkerWatcher 的实现非常简单，因此感兴趣的读者可自行阅读。

9.10.2 CoarseGrainedExecutorBackend 的功能分析

按照本书一往的安排，笔者依然先来介绍 CoarseGrainedExecutorBackend 的属性，这些属性如下。

- rpcEnv：Executor 所需的 RpcEnv。
- driverUrl：Driver 的 Spark 格式的 URL。例如，spark://CoarseGrainedScheduler@192.168.0.111:53777。
- executorId：Master 分配给 Executor 的身份标识。

- hostname：主机名。
- cores：分配给 Executor 的内核数。
- userClassPath：用户指定的类路径。
- env：Executor 所需的 SparkEnv。
- stopping：标记 CoarseGrainedExecutorBackend 是否正在停止。
- executor：Executor 实例。
- driver：DriverEndpoint 的 RpcEndpointRef。
- ser：Executor 所需的 SparkEnv 中的 closureSerializer 的实例。

有了对以上属性的了解，现在可以展开对 CoarseGrainedExecutorBackend 的核心功能的分析了。

1. CoarseGrainedExecutorBackend 的 onStart 方法

根据之前对 CoarseGrainedExecutorBackend 的伴生对象的 run 方法的分析，我们知道，run 方法中创建了 CoarseGrainedExecutorBackend，并将它注册到 Executor 自身的 RpcEnv 中。在注册到 RpcEnv 的过程中会触发对 CoarseGrainedExecutorBackend 的 onStart 方法的调用，其实现如下。

```
override def onStart() {
  logInfo("Connecting to driver: " + driverUrl)
  rpcEnv.asyncSetupEndpointRefByURI(driverUrl).flatMap { ref =>
    driver = Some(ref)
    ref.ask[Boolean](RegisterExecutor(executorId, self, hostname, cores, extract-
        LogUrls))
  }(ThreadUtils.sameThread).onComplete {
    case Success(msg) =>
    case Failure(e) =>
      exitExecutor(1, s"Cannot register with driver: $driverUrl", e, notifyDriver =
          false)
  }(ThreadUtils.sameThread)
}
```

根据上述代码，onStart 方法中向 DriverEndpoint 发送 RegisterExecutor 消息注册 Executor。DriverEndpoint 对 RegisterExecutor 消息的处理已经在 9.9.2 节介绍过。

2. CoarseGrainedExecutorBackend 的 receive 方法

CoarseGrainedExecutorBackend 的 receive 方法中实现了对 RegisteredExecutor、LaunchTask、KillTask 等消息的处理，如代码清单 9-91 所示。

代码清单9-91 CoarseGrainedExecutorBackend的receive方法

```
override def receive: PartialFunction[Any, Unit] = {
  case RegisteredExecutor =>
    logInfo("Successfully registered with driver")
    try {
      executor = new Executor(executorId, hostname, env, userClassPath, isLocal =
          false)
```

```
      } catch {
        case NonFatal(e) =>
          exitExecutor(1, "Unable to create executor due to " + e.getMessage, e)
      }

    case RegisterExecutorFailed(message) =>
      exitExecutor(1, "Slave registration failed: " + message)

    case LaunchTask(data) =>
      if (executor == null) {
        exitExecutor(1, "Received LaunchTask command but executor was null")
      } else {
        val taskDesc = ser.deserialize[TaskDescription](data.value)
        logInfo("Got assigned task " + taskDesc.taskId)
        executor.launchTask(this, taskId = taskDesc.taskId, attemptNumber = task-
            Desc.attemptNumber,
          taskDesc.name, taskDesc.serializedTask)
      }

    case KillTask(taskId, _, interruptThread) =>
      if (executor == null) {
        exitExecutor(1, "Received KillTask command but executor was null")
      } else {
        executor.killTask(taskId, interruptThread)
      }

    case StopExecutor =>
      stopping.set(true)
      logInfo("Driver commanded a shutdown")
      self.send(Shutdown)

    case Shutdown =>
      stopping.set(true)
      new Thread("CoarseGrainedExecutorBackend-stop-executor") {
        override def run(): Unit = {
          executor.stop()
        }
      }.start()
  }
```

3. CoarseGrainedExecutorBackend 的 statusUpdate 方法

在 Standalone 模式下，由于 CoarseGrainedExecutorBackend 与 Executor 在同一个进程内，所以它对 Task 的状态能够准确实时地捕获。利用这一天生优势，CoarseGrainedExecutorBackend 可以将 Task 的状态发送给 DriverEndpoint，以便对 Task 的状态进行更新。CoarseGrainedExecutorBackend 的 statusUpdate 方法实现了这一功能，如代码清单 9-92 所示。

代码清单9-92 CoarseGrainedExecutorBackend的statusUpdate方法

```
override def statusUpdate(taskId: Long, state: TaskState, data: ByteBuffer) {
  val msg = StatusUpdate(executorId, taskId, state, data)
  driver match {
    case Some(driverRef) => driverRef.send(msg)
    case None => logWarning(s"Drop $msg because has not yet connected to driver")
```

```
    }
  }
```

9.11 local-cluster 部署模式

local-cluster 是一种伪集群部署模式，Driver、Master 和 Worker 在同一个 JVM 进程内，可以存在多个 Worker，每个 Worker 会有多个 Executor，但这些 Executor 都独自存在于一个 JVM 进程内。除了这些，它与 local 部署模式的区别如下。

- 使用 LocalSparkCluster 启动集群。
- StandaloneSchedulerBackend 的启动过程不同。
- AppClient 的启动与调度。
- local-cluster 模式的任务执行。

在第 7 章，笔者曾讲解了 TaskSchedulerImpl 和 local 模式下 SchedulerBackend 的实现 LocalSchedulerBackend。本节我们设置 master 为 local-cluster[2,1,1024]，那么在创建 TaskSchedulerImpl 时（见代码清单 4-18）就会匹配 local-cluster 模式。local-cluster[2,1,1024] 中的 2 指定了 Worker 的数量（numSlaves），1 指定了每个 Worker 占用的 CPU 核数（coresPerSlave），1024 指定的是每个 Worker 占用的内存大小（memoryPerSlave）。memoryPerSlave 必须大于等于 executorMemory（即 SparkContext 的 _executorMemory 属性），因为 Worker 的内存大小包括 Executor 占用的内存。

与 local 模式不同的是，local-cluster 除 TaskSchedulerImpl 外，还创建了 LocalSparkCluster，LocalSparkCluster 的 start 方法用于启动集群。local-cluster 模式中，ExecutorBackend 的实现类是 CoarseGrainedExecutorBackend，SchedulerBackend 的实现类是 StandaloneSchedulerBackend。


注意 给 StandaloneSchedulerBackend 的 shutdownCallback 绑定 LocalSparkCluster 的 stop 方法，用于当 Driver 关闭时关闭集群。这仅限于 local-cluster 模式。

9.11.1 启动本地集群

LocalSparkCluster 实现了本地集群。LocalSparkCluster 包含以下属性。

- numWorkers：Worker 的数量。
- coresPerWorker：每个 Worker 的内核数。
- memoryPerWorker：每个 Worker 的内存大小。
- conf：即 SparkConf。
- localHostname：本地主机名。通过调用 Utils 工具类的 localHostName 方法获得。
- masterRpcEnvs：用于缓存所有 Master 的 RpcEnv。
- workerRpcEnvs：维护所有 Worker 的 RpcEnv。

- masterWebUIPort：Master 的 WebUI 的端口。

有了对 LocalSparkCluster 中定义的成员属性的了解，现在来看看如何启动本地集群。LocalSparkCluster 的 start 方法（见代码清单 9-93）用于创建、启动 Master 的 RpcEnv 与多个 Worker 的 RpcEnv。

代码清单9-93 LocalSparkCluster的start方法

```
def start(): Array[String] = {
  logInfo("Starting a local Spark cluster with " + numWorkers + " workers.")

  val _conf = conf.clone()
    .setIfMissing("spark.master.rest.enabled", "false")
    .set("spark.shuffle.service.enabled", "false")

  /* 启动Master */
  val (rpcEnv, webUiPort, _) = Master.startRpcEnvAndEndpoint(localHostname, 0, 0, _
     conf)
  masterWebUIPort = webUiPort
  masterRpcEnvs += rpcEnv
  val masterUrl = "spark://" + Utils.localHostNameForURI() + ":" + rpcEnv.
     address.port
  val masters = Array(masterUrl)

  for (workerNum <- 1 to numWorkers) { /* 启动多个Worker */
    val workerEnv = Worker.startRpcEnvAndEndpoint(localHostname, 0, 0, cores-
       PerWorker,
      memoryPerWorker, masters, null, Some(workerNum), _conf)
    workerRpcEnvs += workerEnv
  }

  masters
}
```

根据代码清单 9-93，启动本地 Spark 集群的步骤如下。

1）在本地模式下通过将 spark.master.rest.enabled 和 spark.shuffle.service.enabled 设置为 false，使得 REST 服务不可用。

2）调用 Master 的伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-24）启动 Master，然后将 Master 的 WebUI 的端口保存到 masterWebUIPort 属性，并将 Master 的 RpcEnv 放入 masterRpcEnvs 缓存，最后创建 Master 的 URL。

3）按照 numWorkers 指定的 Worker 数量，调用 Worker 的伴生对象的 startRpcEnvAndEndpoint 方法（见代码清单 9-50）启动多个 Worker，并将每个 Worker 的 RpcEnv 放入 workerRpcEnvs 缓存。

4）返回 Master 的 URL。

此外，LocalSparkCluster 还提供了 stop 方法用于关闭、清理 Master 的 RpcEnv 与多个 Worker 的 RpcEnv。由于 stop 方法的实现非常简单，因此留给感兴趣的读者自行阅读。

9.11.2 local-cluster 部署模式的启动过程

有了本章对 Master、Worker、StandaloneAppClient、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 local-cluster 部署模式的启动过程，如图 9-21 所示。

这里对图 9-21 中的各个序号进行说明。

序号①：根据 SparkContext 的 createTaskScheduler 方法（见代码清单 4-18）匹配 local-cluster 模式的代码，首先调用 LocalSparkCluster 的 start 方法启动本地集群。

序号②：根据 9.11.1 节的介绍，LocalSparkCluster 的 start 方法首先创建并启动 Master。

序号③：启动 Master 的过程（详见 9.6.1 节）中将创建领导选举代理（LeaderElection-Agent）。

序号④：由于 local-cluster 模式下只有一个 Master，所以无论采用哪种领导选举代理，都将选择 Master 作为激活的 Master。

序号⑤：LocalSparkCluster 的 start 方法之后会根据 local-cluster[numSlaves, coresPer-Slave, memoryPerSlave] 指定的 numSlaves 的大小，创建并启动一到多个 Worker。

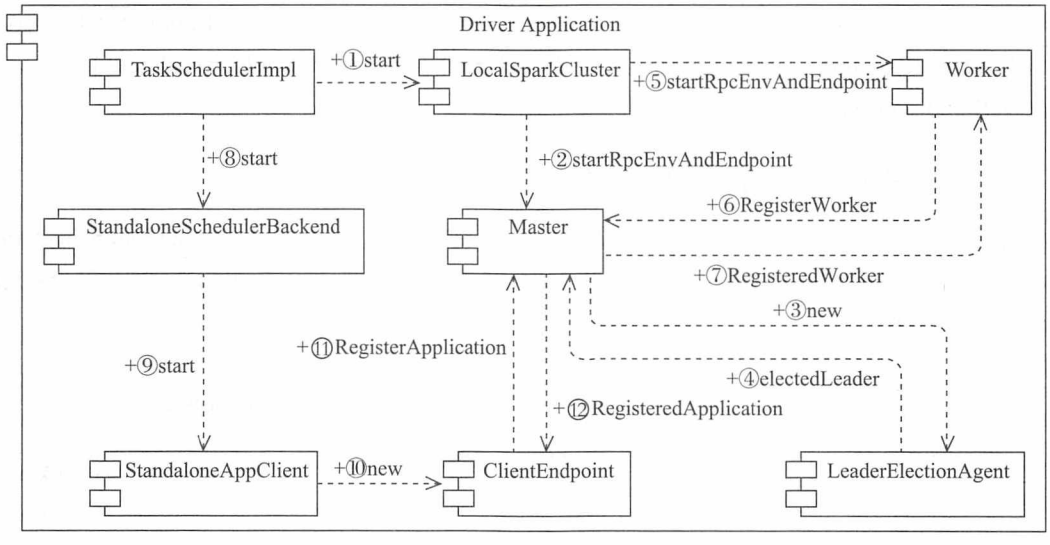


图 9-21 local-cluster 部署模式的启动过程

序号⑥：在启动 Worker 的过程中将向 Master 发送 RegisterWorker 消息（见 9.7.2 节）。

序号⑦：Master 在接收到 RegisterWorker 消息后对 WorkerInfo 进行注册，WorkerInfo 注册成功后，Master 向 Worker 回复 RegisteredWorker 消息（见 9.6.5 节）。Worker 收到 RegisteredWorker 消息后，将向 Master 发送 WorkerLatestState 消息，以便把 Worker 的最新状态汇报给 Master。根据 9.6.6 节的内容，我们知道 Master 接收到 WorkerLatestState 消息后，将对 Executor 和 Driver 进行匹配，对于不匹配的 Executor 和 Driver，将通知 Worker “杀死”。


注意 在通过 LocalSparkCluster 启动本地集群后，会对 TaskSchedulerImpl 进行初始化。根据 7.10.2 节的内容，我们知道 TaskSchedulerImpl 的 initialize 方法会将参数传递的 StandaloneSchedulerBackend 保存到 TaskSchedulerImpl 的 backend 属性。

序号⑧：SparkContext 创建完 TaskSchedulerImpl 后，将启动 TaskSchedulerImpl（见代码清单 4-19）。根据 7.10.3 节对启动 TaskSchedulerImpl 的分析，TaskSchedulerImpl 的 start 方法一开始就会调用 StandaloneSchedulerBackend 的 start 方法。

序号⑨：通过 9.9.3 节对 StandaloneSchedulerBackend 的启动分析，StandaloneSchedulerBackend 的 start 方法中会创建并启动 StandaloneAppClient。

序号⑩：StandaloneAppClient 的 start 方法会创建 ClientEndpoint 并将 ClientEndpoint 注册到 SparkContext 的 SparkEnv 的 RpcEnv 中，进而引起 ClientEndpoint 的启动。

序号⑪：9.8.1 节详细介绍了 ClientEndpoint 的启动，ClientEndpoint 在启动时会向每个 Master 发送 RegisterApplication 消息。

序号⑫：根据 9.6.8 节的介绍，Master 收到 RegisterApplication 消息并将 Application 注册成功后，会向 DriverEndpoint 发送 RegisteredApplication 消息。ClientEndpoint 的 receive 方法接收到 RegisteredApplication 消息后更新 ClientEndpoint 的 appId、registered、master 属性及 StandaloneSchedulerBackend 的 appId 属性。

9.11.3 local-cluster 部署模式下 Executor 的分配过程

有了本章对 Master、Worker、StandaloneAppClient、DriverEndpoint、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 local-cluster 部署模式下的 Executor 资源分配过程，如图 9-22 所示。

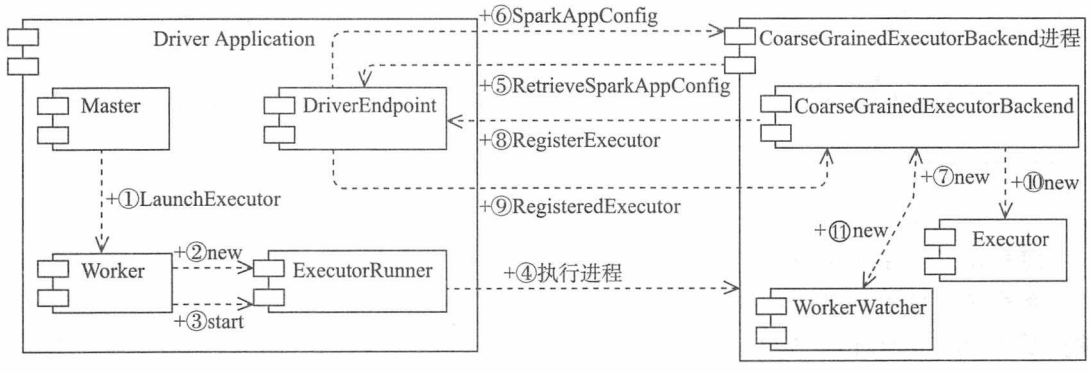


图 9-22 local-cluster 部署模式下的 Executor 资源分配过程

这里对图 9-22 中的各个序号进行说明。

序号①：Master 的 schedule 方法在对资源调度时将调用 startExecutorsOnWorkers 方

法对 Executor 的资源进行分配和调度，然后通过向 Worker 发送 LaunchExecutor 消息运行 Executor。

序号②：Worker 接收到 LaunchExecutor 消息后将创建 ExecutorRunner。

序号③：Worker 创建完 ExecutorRunner 后，将调用 ExecutorRunner 的 start 方法。

序号④：ExecutorRunner 的 start 方法主要创建了线程 workerThread，线程 workerThread 将构建并启动 ProcessBuilder。由于 ProcessBuilder 的命令中指定了 Java 进程的主方法是 CoarseGrainedExecutorBackend 的伴生对象的 main 函数，因此将启动执行 CoarseGrainedExecutorBackend 的 main 函数的进程。

序号⑤：执行 CoarseGrainedExecutorBackend 的 main 函数的进程首先创建名为 driverPropsFetcher 的 RpcEnv，然后向 DriverEndpoint 发送 RetrieveSparkAppConfig 消息。

序号⑥：DriverEndpoint 接收到 RetrieveSparkAppConfig 消息后，将向执行 CoarseGrainedExecutorBackend 的 main 函数的进程回复 SparkAppConfig 消息。SparkAppConfig 消息携带着 Spark 属性信息和分配给 Executor 的密钥。

序号⑦：CoarseGrainedExecutorBackend 的 main 函数的进程收到 SparkAppConfig 消息后，将首先创建 SparkConf 和 SparkEnv，然后创建 CoarseGrainedExecutorBackend 实例并注册到 Executor 自身的 SparkEnv 的 RpcEnv。

序号⑧：将 CoarseGrainedExecutorBackend 实例注册到 Executor 自身的 SparkEnv 的 RpcEnv 时，会触发对 CoarseGrainedExecutorBackend 的 onStart 方法的调用，进而向 DriverEndpoint 发送 RegisterExecutor 消息以注册 Executor。

序号⑨：DriverEndpoint 接收到 RegisterExecutor 消息对 Executor 注册成功后，将向 CoarseGrainedExecutorBackend 实例回复 RegisteredExecutor 消息。

序号⑩：CoarseGrainedExecutorBackend 实例接收到 RegisteredExecutor 消息后将创建 Executor。

序号⑪：CoarseGrainedExecutorBackend 的 main 函数的进程收到 SparkAppConfig 消息后，除了创建 CoarseGrainedExecutorBackend 实例，还将创建 WorkerWatcher，以保证执行 CoarseGrainedExecutorBackend 的 main 函数的进程在与 Worker 断开连接或发生网络错误的情况下退出。

9.11.4 local-cluster 部署模式下的任务提交执行过程

有了本章对 Master、Worker、StandaloneAppClient、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 local-cluster 部署模式下的任务提交执行过程，如图 9-23 所示。

这里对图 9-23 中的各个序号进行说明。

序号①：TaskSchedulerImpl 的 submitTasks 方法（见代码清单 7-99）提交 Task 的最后会调用 StandaloneSchedulerBackend 的父类 CoarseGrainedSchedulerBackend 的 reviveOffers

方法。

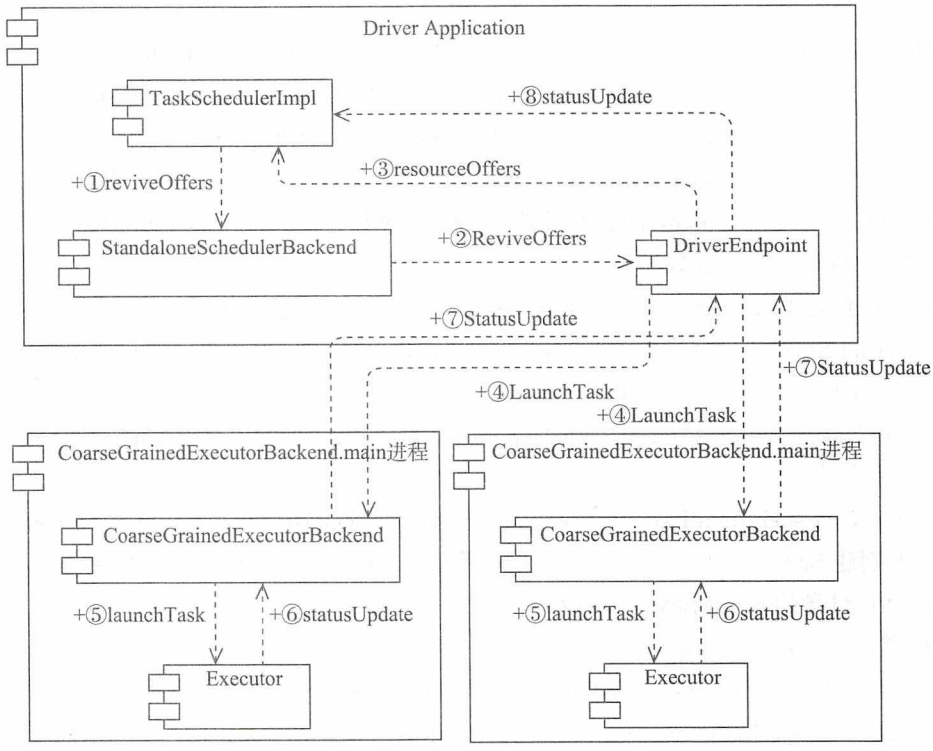


图 9-23　local-cluster 部署模式下的任务提交执行过程

序号②：CoarseGrainedSchedulerBackend 的 reviveOffers 方法只是向 DriverEndpoint 发送 ReviveOffers 消息。

序号③：DriverEndpoint 接收到 ReviveOffers 消息后，首先调用 TaskSchedulerImpl 的 resourceOffers 方法给 Task 分配资源。

序号④：给 Task 分配好资源后，向 CoarseGrainedExecutorBackend 实例发送 LaunchTask 消息。

序号⑤：CoarseGrainedExecutorBackend 实例接收到 LaunchTask 消息后，调用 Executor 的 launchTask 方法运行 Task。

序号⑥：Executor 在运行 Task 的过程中将不断调用 CoarseGrainedExecutorBackend 的 statusUpdate 方法更新 Task 状态。

序号⑦：CoarseGrainedExecutorBackend 的 statusUpdate 方法将向 DriverEndpoint 发送 StatusUpdate 消息。

序号⑧：DriverEndpoint 接收到 StatusUpdate 消息后调用 TaskSchedulerImpl 的 statusUpdate 方法更新 Task 状态。

9.12 Standalone 部署模式

local 模式只有 Driver 和 Executor，且在同一个 JVM 进程内。local-cluster 模式的 Driver、Master、Worker 也都位于同一个 JVM 进程内部。所以 local 模式和 local-cluster 模式便于开发、测试，也便于源码阅读与调试，但是却不适合在生产环境使用。Standalone 部署模式有哪些特点呢？

- Driver 是单独的进程，可以存在于集群之内，也可以存在于集群之外，对 Spark Application 的执行进行驱动。
- Master 是单独的进程，甚至应该在单独的机器节点上。Master 有多个，但同时最多只有一个处于激活状态。
- Worker 是单独的进程，也推荐在单独的机器节点上部署。

我们以 Linux 环境下启动 Standalone 模式为例。启动 Standalone 模式需要保证一定的顺序，即先启动 Master，再逐个启动 Worker。${SPARK_HOME}/sbin/start-master.sh 脚本将以 org.apache.spark.deploy.master.Master 作为主类（已在 9.6.1 节详细介绍），启动 JVM 进程。启动 Master 的命令如图 9-24 所示。

Master 进程的默认端口是 7077，WEBUI 的端口是 8080。可以用 jps 命令查看 Master 进程的执行参数，输出如图 9-25 所示。

```
EBJ1266:sbin user$ ./start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /Users/user/install/spark-2.0.0-bin-hadoop
y.master.Master-1-EBJ1266.local.out
```

图 9-24 启动 Master 进程

```
EBJ1266:sbin user$ jps -lmv
1154 org.apache.spark.deploy.master.Master --host EBJ1266.local --port 7077 --webui-port 8080 -Xmx1g
```

图 9-25 Master 进程的执行参数

启动完 Master，就可以启动 Worker 了。${SPARK_HOME}/sbin/start-slaves.sh 或 ${SPARK_HOME}/sbin/start-slave.sh 都能启动 Worker，区别在于 start-slaves.sh 可以批量启动多个 Worker。为简单起见，笔者使用 start-slave.sh。start-slave.sh 脚本以 org.apache.spark.deploy.worker.Worker 为主类（已在 9.7.1 节详细介绍），启动 JVM 进程。我们指定 Master 的连接地址，然后启动 Worker 的命令如图 9-26 所示。

```
EBJ1266:sbin user$ ./start-slave.sh spark://127.0.0.1:7077
starting org.apache.spark.deploy.worker.Worker, logging to /Users/user/install/spark-2.0.0-bin-hadoop2.6/logs/
y.worker.Worker-1-EBJ1266.local.out
```

图 9-26 启动 Worker 进程

我们同样用 jps –lmv 命令查看 Woker 进程的执行参数，如图 9-27 所示。

```
1308 org.apache.spark.deploy.worker.Worker --webui-port 8081 spark://127.0.0.1:7077 -Xmx1g
```

图 9-27 Worker 进程的执行参数

有关 Master 和 Worker 的实现分析，前文都已经详细介绍，本节我们设置 master 为 spark://127.0.0.1:7077，那么在创建 TaskSchedulerImpl 时（见代码清单 4-18）就会匹配 Standalone 模式。

Standalone 模式中，ExecutorBackend 的实现类是 CoarseGrainedExecutorBackend，SchedulerBackend 的实现类是 StandaloneSchedulerBackend。由于 Standalone 模式中的各个 Master 和 Worker 都是单独的进程，因此不再需要 local-cluster 模式中的 LocalSparkCluster 来启动位于 JVM 进程内的 Master 和 Worker。本章已经详细分析了 Master 和 Worker 的实现，下面将展示 Standalone 模式下最主要的工作原理。由于 Standalone 部署模式下的任务提交执行过程与 local-cluster 模式下的相同，所以不再展示。

9.12.1 Standalone 部署模式的启动过程

有了本章对 Master、Worker、StandaloneAppClient、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 Standalone 部署模式的启动过程。Standalone 部署模式能够选用的领导选举代理只能是 ZooKeeperLeaderElectionAgent，本节以 2 个 Worker、2 个 Master、1 个 Driver application 的情况，展示整个 Standalone 部署模式的启动过程，如图 9-28 所示。

这里对图 9-28 中的各个序号进行说明。

序号①：Standalone 部署模式中首先会启动一到多个 Master.main 函数的进程。每个 Master.main 进程中都会创建 Master 实例并注册到 Master 自己的 RpcEnv 中，在启动 Master 实例的过程中会创建 ZooKeeperLeaderElectionAgent 实例。

序号②：每个 Master 实例对应的 ZooKeeperLeaderElectionAgent 实例都会参与 ZooKeeper 的领导选举，最终确定正式的领导者并调用 ZooKeeperLeaderElectionAgent 的 isLeader 方法。isLeader 方法将会调用 Master 的 electedLeader 方法将 Master 选举为领导，Master 的状态被设置为激活。对于没有被选举为领导的 Master，会调用 ZooKeeperLeaderElectionAgent 的 notLeader 方法。notLeader 方法会将领导者的信息告知支持者。

序号③：Standalone 部署模式中会启动一到多个 Worker.main 函数的进程。每个 Worker.main 进程中都会创建 Worker 实例并注册到 Worker 自己的 RpcEnv 中，在启动 Worker 实例的过程中会向 Master 实例发送 RegisterWorker 消息。

序号④：Master 接收到 RegisterWorker 消息后对 WorkerInfo 进行注册。WorkerInfo 注册成功后，Master 向 Worker 回复 RegisteredWorker 消息（见 9.6.5 节）。Worker 收到 RegisteredWorker 消息后，将向 Master 发送 WorkerLatestState 消息，以便把 Worker 的最新状态汇报给 Master。根据 9.6.6 节的内容，我们知道 Master 接收到 WorkerLatestState 消息后，将对 Executor 和 Driver 进行匹配，对于不匹配的 Executor 和 Driver，将通知 Worker “杀死”。

序号⑤：SparkContext 创建完 TaskSchedulerImpl 后，将启动 TaskSchedulerImpl（见代码清单 4-19）。根据 7.10.3 节对启动 TaskSchedulerImpl 的分析，TaskSchedulerImpl 的 start

方法一开始就会调用 StandaloneSchedulerBackend 的 start 方法。

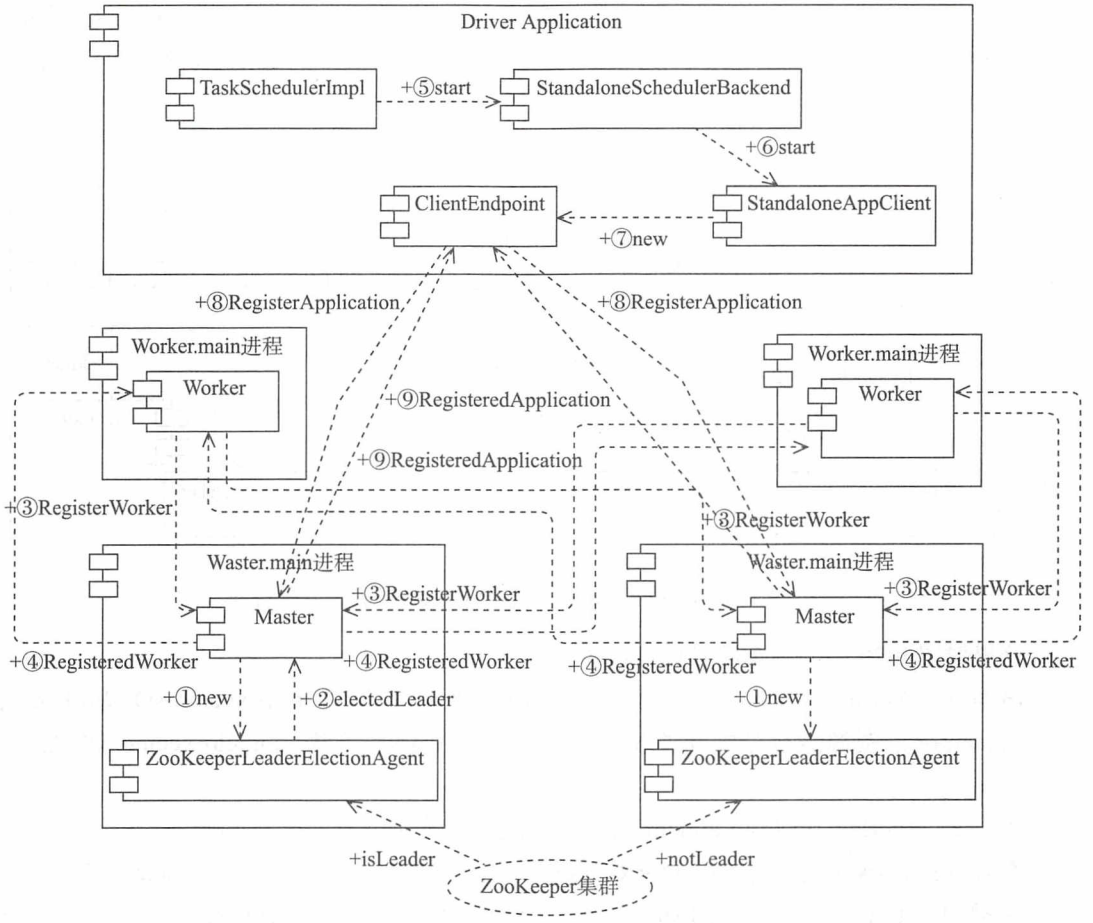


图 9-28 Standalone 部署模式的启动过程

序号⑥：通过 9.9.3 节对 StandaloneSchedulerBackend 的启动分析，StandaloneSchedulerBackend 的 start 方法中会创建并启动 StandaloneAppClient。

序号⑦：StandaloneAppClient 的 start 方法会创建 ClientEndpoint 并将 ClientEndpoint 注册到 SparkContext 的 SparkEnv 的 RpcEnv 中，进而引起 ClientEndpoint 的启动。

序号⑧：9.8.1 节详细介绍了 ClientEndpoint 的启动，ClientEndpoint 在启动时会向每个 Master 发送 RegisterApplication 消息。

序号⑨：根据 9.6.8 节的介绍，Master 收到 RegisterApplication 消息并注册成功后，会向 DriverEndpoint 发送 RegisteredApplication 消息。ClientEndpoint 的 receive 方法接收到 RegisteredApplication 消息（见代码清单 9-70）后，更新 ClientEndpoint 的 appId、registered、master 属性及 StandaloneSchedulerBackend 的 appId 属性。

9.12.2 Standalone 部署模式下 Executor 的分配过程

有了本章对 Master、Worker、StandaloneAppClient、DriverEndpoint、StandaloneSchedulerBackend 等内容的分析，现在可以整理出 Standalone 部署模式下的 Executor 资源分配过程，如图 9-29 所示。

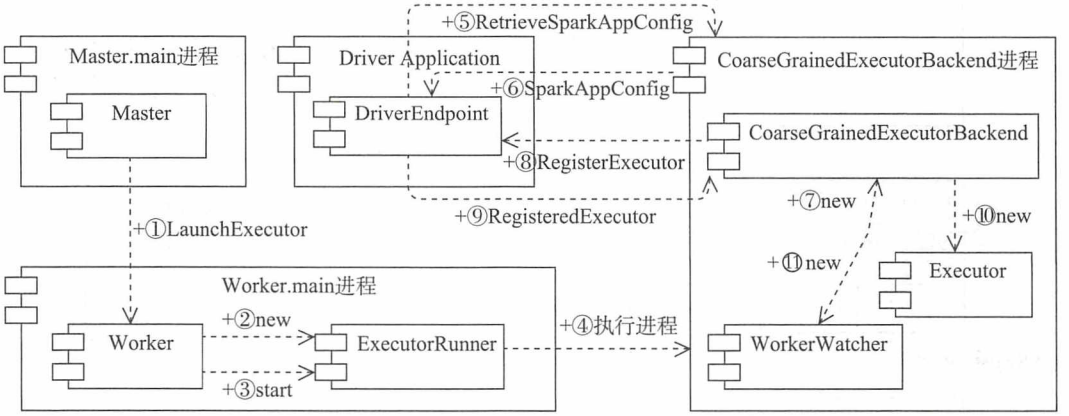


图 9-29 Standalone 部署模式下的 Executor 资源分配过程

这里对图 9-29 中的各个序号进行说明。

序号①：Master 的 schedule 方法在对资源调度时将调用 startExecutorsOnWorkers 方法对 Executor 的资源进行分配和调度，然后通过向 Worker 发送 LaunchExecutor 消息运行 Executor。

序号②：Worker 接收到 LaunchExecutor 消息后将创建 ExecutorRunner。

序号③：Worker 创建完 ExecutorRunner 后，将调用 ExecutorRunner 的 start 方法。

序号④：ExecutorRunner 的 start 方法主要创建了线程 workerThread，线程 workerThread 将构建并启动 ProcessBuilder。由于 ProcessBuilder 的命令中指定了 Java 进程的主方法是 CoarseGrainedExecutorBackend 的伴生对象的 main 函数，因此将启动执行 CoarseGrainedExecutorBackend 的 main 函数的进程。

序号⑤：执行 CoarseGrainedExecutorBackend 的 main 函数的进程首先创建名为 driverPropsFetcher 的 RpcEnv，然后向 DriverEndpoint 发送 RetrieveSparkAppConfig 消息。

序号⑥：DriverEndpoint 接收到 RetrieveSparkAppConfig 消息后，将向执行 CoarseGrainedExecutorBackend 的 main 函数的进程回复 SparkAppConfig 消息。SparkAppConfig 消息携带着 Spark 属性信息和分配给 Executor 的密钥。

序号⑦：CoarseGrainedExecutorBackend 的 main 函数的进程收到 SparkAppConfig 消息后，将首先创建 SparkConf 和 SparkEnv，然后创建 CoarseGrainedExecutorBackend 实例并注册到 Executor 自身的 SparkEnv 的 RpcEnv。

序号⑧：将 CoarseGrainedExecutorBackend 实例注册到 Executor 自身的 SparkEnv 的

RpcEnv 时，会触发对 CoarseGrainedExecutorBackend 的 onStart 方法的调用，进而向 DriverEndpoint 发送 RegisterExecutor 消息以注册 Executor。

序号⑨：DriverEndpoint 接收到 RegisterExecutor 消息对 Executor 注册成功后，将向 CoarseGrainedExecutorBackend 实例回复 RegisteredExecutor 消息。

序号⑩：CoarseGrainedExecutorBackend 实例接收到 RegisteredExecutor 消息后将创建 Executor。

序号⑪：CoarseGrainedExecutorBackend 的 main 函数的进程收到 SparkAppConfig 消息后，除了创建 CoarseGrainedExecutorBackend 实例，还将创建 WorkerWatcher，以保证执行 CoarseGrainedExecutorBackend 的 main 函数的进程在与 Worker 断开连接或发生网络错误的情况下退出。

9.12.3 Standalone 部署模式的资源回收

除了流式计算，大多数情况下的任务在运行一段时间之后都会完成，那么 Application 占用的资源应该被回收。Master 和 Executor 是如何感知到 Application 的退出呢？

Spark 中有两种处理方式：一种是离别之前先打声招呼，一种是不辞而别。

1. 离别之前先打声招呼

这种方式很好理解，SparkContext 提供了 stop 方法用于告别，即停止各种服务和回收资源。SparkContext 的 stop 方法中有一段调用 DagScheduler 的 stop 方法的代码。

```
if (_dagScheduler != null) {
  Utils.tryLogNonFatalError {
    _dagScheduler.stop()
  }
  _dagScheduler = null
}
```

DagScheduler 的 stop 方法中会调用 TaskSchedulerImpl 的 stop 方法。

```
taskScheduler.stop()
```

TaskSchedulerImpl 的 stop 方法中会调用 StandaloneSchedulerBackend 的 stop 方法。

```
  if (backend != null) {
    backend.stop()
}
```

根据 9.9.4 节的介绍，StandaloneSchedulerBackend 的 stop 方法将引起向 DriverEnd-point 发送 StopExecutors 消息。DriverEndpoint 接收到 StopExecutors 消息后，向每个 CoarseGrainedExecutorBackend 实例发送 StopExecutor 消息。CoarseGrainedExecutorBackend 的 receive 方法（见代码清单 9-91）接收到 StopExecutor 消息后，将向 CoarseGrainedExecutorBackend 自己发送 Shutdown 消息。CoarseGrainedExecutorBackend 接收到 Shutdown 消息，将启动一个线程用于调用 Executor 的 stop 方法（此方法的实现非常简单，留给感兴趣的读

者自行阅读）停止 Executor。

2. 不辞而别

上面的分析发现 Application 只记得跟 Executor 打声招呼，却忘记了 Master。这该怎么办？下面将分析具体的解决方案。

在 5.3.5 节介绍 NettyRpcHandler 时，提到过 NettyRpcHandler 除实现了 RpcHandler 的两个 receive 方法，还实现了 exceptionCaught、channelActive 与 channelInactive 等。exceptionCaught 方法将会向 Inbox 中投递 RemoteProcessConnectionError 消息。channelActive 将会向 Inbox 中投递 RemoteProcessConnected 消息。channelInactive 将会向 Inbox 中投递 RemoteProcessDisconnected 消息。Inbox 的 process 方法（见代码清单 5-7）在处理 RemoteProcessConnected、RemoteProcessDisconnected、RemoteProcessConnectionError 三个消息时，将分别调用 RpcEndpoint 的 onConnected、onDisconnected、onNetworkError 三个方法。

Application 一旦退出，那么 Master 的 onDisconnected 方法将被调用。Master 的 onDisconnected 方法的实现如代码清单 9-94 所示。

代码清单9-94　Master的onDisconnected方法

```
override def onDisconnected(address: RpcAddress): Unit = {
  logInfo(s"$address got disassociated, removing it.")
  addressToWorker.get(address).foreach(removeWorker)
  addressToApp.get(address).foreach(finishApplication)
  if (state == RecoveryState.RECOVERING && canCompleteRecovery) { completeRecov-
      ery() }
}
```

根据代码清单 9-94，Master 将调用 finishApplication 方法处理 Application。finishApplication 方法的实现如下。

```
private def finishApplication(app: ApplicationInfo) {
  removeApplication(app, ApplicationState.FINISHED)
}
```

可以看到 finishApplication 方法中调用了 removeApplication 方法（见代码清单 9-49）移除 Application 及分配给 Application 的 Executor。

9.12.4　Standalone 部署模式的容错机制

在分布式系统中，由于机器数量众多，导致机器发生故障的概率很高，所以在设计任何分布式系统时都应考虑容错。本节主要针对 Standalone 部署模式的容错能力进行分析。

1. Executor 异常退出

如果 Executor 异常退出，那么势必导致在此 Executor 上执行的任务无法运行。曾经在 9.7.6 节介绍 ExecutorRunner 的 fetchAndRunExecutor 方法时，介绍过 CoarseGrainedExecutorBackend 进程退出后，ExecutorRunner 会向 Worker 发送 ExecutorStateChanged 消

仅供非商业用途或交流学习使用

息。根据 9.7.7 节的介绍，Worker 收到 ExecutorStateChanged 消息后将向 Master 转发 ExecutorStateChanged 消息。根据 9.6.10 节的介绍，Master 收到 ExecutorStateChanged 消息后对退出状态的处理步骤如下。

1）找到使用此 Executor 的 Application 所对应的 ApplicationInfo，以及 Executor 对应的 ExecutorInfo。

2）将 ExecutorInfo 的状态改为 EXITED。

3）EXITED 也属于 Executor 完成状态，所以会将 ExecutorInfo 从 ApplicationInfo 和 WorkerInfo 中移除。

4）由于 Executor 是非正常退出，所以重新调用 schedule 方法给 Application 进行资源调度。

Executor 异常退出的容错处理可以用图 9-30 表示。

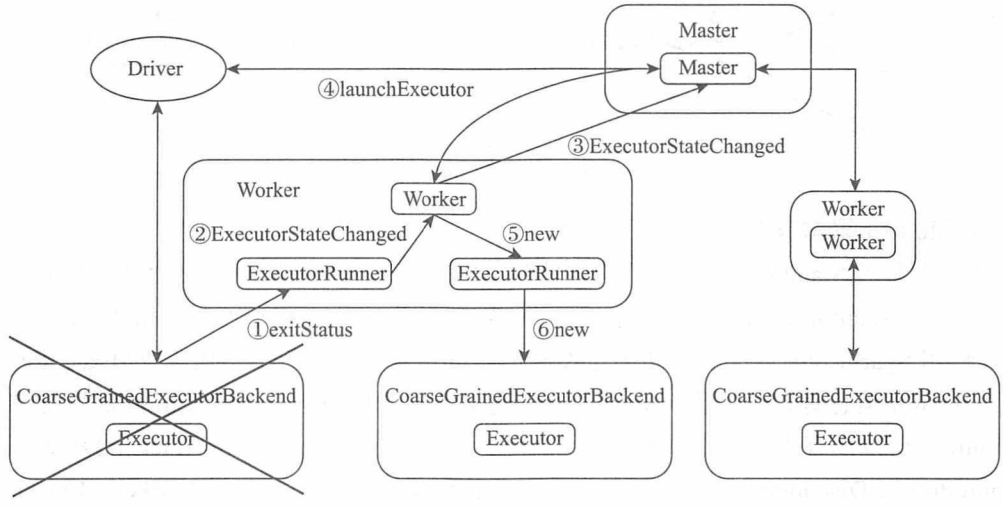


图 9-30 Executor 异常退出的容错处理

第⑥步之后 CoarseGrainedExecutorBackend 将向 Driver 注册 Executor，注册过程已在前面讲过。

2. Worker 异常退出

当 Worker 进程退出时，会发生什么呢？还记得 9.7.6 节中 fetchAndRunExecutor 方法中捕获到异常的处理吗？它将在 Worker 进程退出前调用 killProcess 方法（见代码清单 9-66），主动“杀死”CoarseGrainedExecutorBackend 进程，然后收到进程返回的退出状态（KILLED）后向 Worker 发送 ExecutorStateChanged 消息。由于 Worker 退出了，所以不会有 Heartbeat 消息发送给 Master，所以无法更新 WorkerInfo 的最后心跳时间（lastHeartbeat）。根据 timeOutDeadWorkers 方法（见代码清单 9-28）的实现，Master 会调用 removeWorker 方法（见代码清单 9-48）删除长期失联的 Worker 的 WorkerInfo，并将此 Worker 的所有 Executor

仅供非商业用途或交流学习使用

以 LOST 状态同步更新到它们服务的 Application。最后 Master 还会为 WorkerInfo 所服务的 Application 重新调度，分配到其他 Worker 上。整个过程可以用图 9-31 表示。

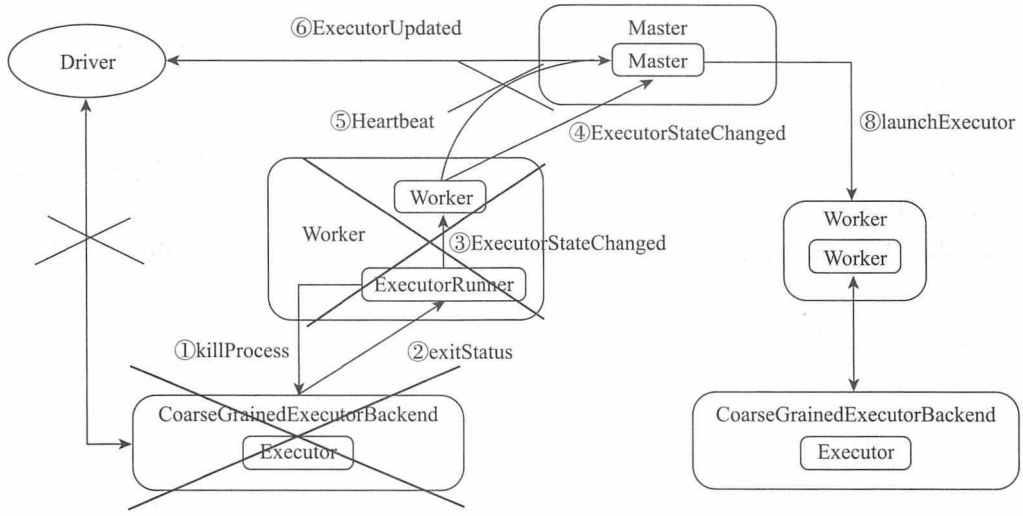


图 9-31 Worker 异常退出的容错处理

3. Master 异常退出

假如我们的 Standlone 模式下的集群只有一个 Master，当 Master 退出时会发生什么呢？

1）当 Executor 上的任务执行完毕，需要对资源回收，按照 9.12.3 节的内容，如果主动调用 SparkContext 的 stop 方法，Application 最终会向给自己服务的 Executor 发送 StopExecutor 消息对资源回收，Master 虽然跑路了，但是 Driver Application 依然持有这些 Executor，所以没有影响。如果不辞而别，那么 CoarseGrainedExecutorBackend 将可以收到 RemoteProcessDisconnected 消息后退出进程。看来 Master 退出，如果 Worker、Executor 正常运行，则对于资源回收没有影响。

2）此时如果再发生 Executor 异常退出问题，Worker 无法通过 ExecutorStateChanged 消息促使 Master 重新将 Application 调度运行 Executor，Application 只能眼巴巴看着自己提交的任务无法执行。而 Executor 占用的资源也无法回收。

3）此时如果又发生 Worker 异常退出问题，那么 Worker 和 Executor 都将停止服务，由于无法通知 Master 重新将 Application 调度到其他 Worker 上，Application 提交的任务也将无法执行。Worker 虽然杀掉了 Executor，但是 Worker 的资源将无法被其他 Application 使用。

4）有新的 Driver 需要提交任务则无法成功。

基于以上分析，发现 Standlone 部署集群只有一个 Master 是万万不可以的，这也称为单点故障问题。解决此问题的常用方式是采用多个 Master，但同时只有一个 Master 负责整个集群的调度、资源管理工作，其他 Master 作为热备。

在 9.4 节和 9.5 节中我们曾经详细分析了故障恢复的持久化引擎（persistenceEngine）

仅供非商业用途或交流学习使用

和领导选举代理（leaderElectionAgent）。在分布式环境下，我们必然要选择 ZooKeeperPersistenceEngine 和 ZooKeeperLeaderElectionAgent。

如果选择 ZooKeeper 作为热备（HA）的方案，在集群启动时需要在 spark-env.sh 增加如下所示的配置。

```
spark.deploy.recoveryMode= ZOOKEEPER
spark.deploy.ZooKeeper.url=cms_zk_server_1:2181, cms_zk_server_2:2181, cms_zk_
    server_3:2181
spark.deploy.ZooKeeper.dir=/usr/zk/persist
```

上面配置中的 spark.deploy.ZooKeeper.url 用于配置使用的 ZooKeeper 集群，理论上需要奇数个 ZooKeeper 节点。spark.deploy.ZooKeeper.dir 是对 Spark 集群状态进行持久化的根目录。

结合 9.6.3 节的内容，使用 ZooKeeper 作为热备的选举方案可以用图 9-32 表示。

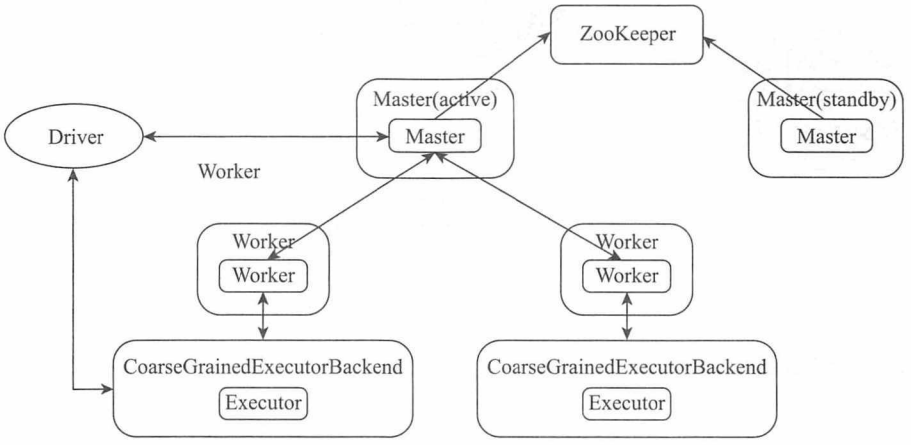


图 9-32 ZooKeeper 热备和选举示意图

ZooKeeper 热备的实现已在 9.6.3 节和 9.7.4 节详细分析过。我们也可以回顾图 9-18。

9.13 其他部署方案

本节介绍 Spark 运行在第三方资源管理集群上的部署方案。Spark 目前支持运行在 YARN 和 Mesos 甚至运行在 MRv1 框架上。对于 MRv1 框架运行 Spark 的内容，有兴趣的读者可以自行研究，本文不做过多介绍。

9.13.1 YARN

在本书第 2 章，笔者曾经简单介绍了 MRv1 的缺陷及 MRv2 的改进。MRv1 的运行时环境由 JobTracker 和 TaskTracker 两类服务组成，JobTracker 负责资源和任务的管理与调度，TaskTracker 负责单个节点的资源管理和任务执行。在 YARN 中，JobTracker 被分为两部分：

仅供非商业用途或交流学习使用

ResourceManager（RM）和 ApplicationMaster（AM）。RM 负责整个集群的资源管理和调度，AM 负责具体应用程序的任务划分、调度等工作，如图 9-33 所示。

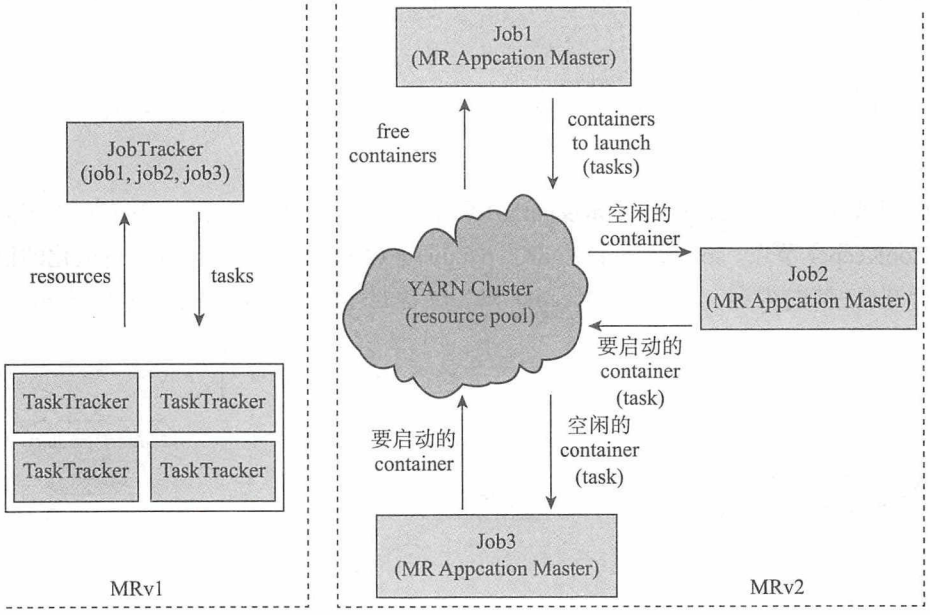


图 9-33　MRv1 与 Yarn 的运行时环境的对比

Yarn 的架构如图 9-34 所示。

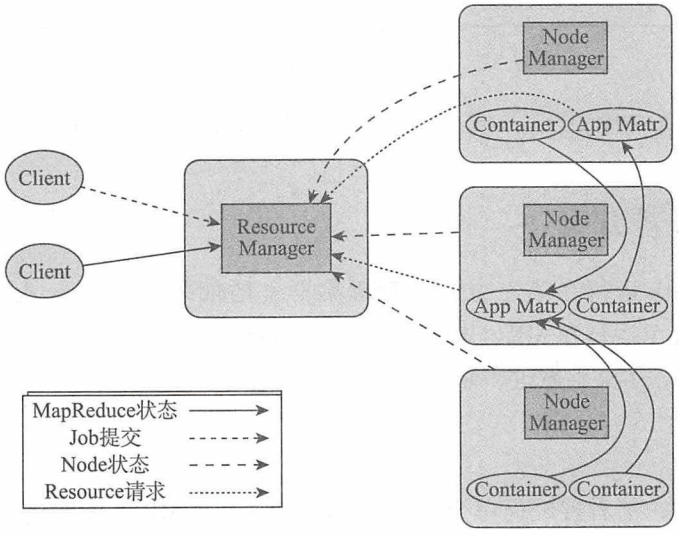


图 9-34　YARN 的架构

YARN 的基本结构如下。

仅供非商业用途或交流学习使用

- ResourceManager（RM）：全局资源管理器，负责整个集群的资源管理与分配。RM由资源调度器和应用程序管理器组成。资源调度器将集群资源分配给各个应用程序，资源的分配单位不再是MRv1中的slot，而是Container。Container是对CPU、内存等资源的封装。应用程序管理器负责管理整个系统的应用程序，如处理程序提交，与资源调度器沟通后为应用程序启动ApplicationMaster等。
- ApplicationMaster（AM）：用户提交的每个应用程序都有一个AM，它会与RM通信获取资源，将作业划分为更细粒度的任务，与NodeManager（NM）通信启动或停止任务，监控失败任务为其重新申请资源后重新启动等。
- NodeManager（NM）：单个节点上的资源与任务管理器，它负责向RM定时汇报本节点的资源使用情况及各个Container的状态，也接受处理AM的启动或停止任务请求。

YARN对于支持的MapReduce框架是可插拔的，Spark对于集群管理器也支持可插拔，两者不谋而合。有关YARN的安装和使用，读者可自行翻阅相关书籍。假设我们已经部署好了YARN集群，根据图9-34所有应用程序需要一个Application Master，这个Application Master位于YARN集群的一个节点上，负责管理资源申请与资源的二级分配调度。Spark也提供了Application Master的实现ApplicationMaster。与YARN集成时整个Spark集群的启动顺序如下。

1）将Spark提供的ApplicationMaster在YARN集群中启动（见代码清单9-95）。

2）ApplicationMaster向ResourceManager申请Container。

3）申请Container成功后，向具体的NodeManager发送指令启动Container。

4）ApplicationMaster启动对各个运行的ContainerExecutor进行监控。



代码清单9-95 ApplicationMaster的main方法

```
def main(args: Array[String]): Unit = {
  SignalUtils.registerLogger(log)
  val amArgs = new ApplicationMasterArguments(args)

  if (amArgs.propertiesFile != null) {
    Utils.getPropertiesFromFile(amArgs.propertiesFile).foreach { case (k, v) =>
      sys.props(k) = v
    }
  }
  SparkHadoopUtil.get.runAsSparkUser { () =>
    master = new ApplicationMaster(amArgs, new YarnRMClient)
    System.exit(master.run())
  }
}
```

ApplicationMaster的registerAM方法（见代码清单9-96）用于向YARN注册AM。

代码清单9-96 ApplicationMaster的registerAM方法

```
private def registerAM(
    _sparkConf: SparkConf,
```

仅供非商业用途或交流学习使用


```
    _rpcEnv: RpcEnv,
    driverRef: RpcEndpointRef,
    uiAddress: String,
    securityMgr: SecurityManager) = {
  val appId = client.getAttemptId().getApplicationId().toString()
  val attemptId = client.getAttemptId().getAttemptId().toString()
  val historyAddress =
    _sparkConf.get(HISTORY_SERVER_ADDRESS)
      .map { text => SparkHadoopUtil.get.substituteHadoopVariables(text, yarn-
          Conf) }
      .map { address => s"${address}${HistoryServer.UI_PATH_PREFIX}/${appId}/
          ${attemptId}" }
      .getOrElse("")

  val driverUrl = RpcEndpointAddress(
    _sparkConf.get("spark.driver.host"),
    _sparkConf.get("spark.driver.port").toInt,
    CoarseGrainedSchedulerBackend.ENDPOINT_NAME).toString

  logInfo {
    val executorMemory = sparkConf.get(EXECUTOR_MEMORY).toInt
    val executorCores = sparkConf.get(EXECUTOR_CORES)
    val dummyRunner = new ExecutorRunnable(None, yarnConf, sparkConf, driverUrl,
      "<executorId>",
      "<hostname>", executorMemory, executorCores, appId, securityMgr, local-
          Resources)
    dummyRunner.launchContextDebugInfo()
  }

  allocator = client.register(driverUrl, driverRef, yarnConf, _sparkConf, uiAdd-
      ress,
    historyAddress, securityMgr, localResources)

  allocator.allocateResources()
  reporterThread = launchReporterThread()
}
```

下面我们设置 master 为 yarn，那么根据代码清单 4-18，在创建 TaskSchedulerImpl 时将无法匹配 local、local-cluster、Standalone 等模式，因而只能匹配最后的 masterUrl。根据代码清单 4-18，首先调用 getClusterManager 方法（见代码清单 9-97）获取集群管理器。

代码清单9-97　SparkContext的getClusterManager方法

```
private def getClusterManager(url: String): Option[ExternalClusterManager] = {
  val loader = Utils.getContextOrSparkClassLoader
  val serviceLoaders =
    ServiceLoader.load(classOf[ExternalClusterManager], loader).asScala.filter(_.
        canCreate(url))
  if (serviceLoaders.size > 1) {
    throw new SparkException(
      s"Multiple external cluster managers registered for the url $url: $service-
          Loaders")
  }
  serviceLoaders.headOption
```

仅供非商业用途或交流学习使用


```
}
```

根据上述代码，getClusterManager方法将通过类加载器加载所有实现了特质ExternalClusterManager的类，并调用实现的canCreate方法进行过滤。此时由于YarnClusterManager的canCreate方法将返回true，因此获得的外部集群管理器就是YarnClusterManager。YarnClusterManager的canCreate方法的实现如下。

```
override def canCreate(masterURL: String): Boolean = {
  masterURL == "yarn"
}
```

根据代码清单4-18，在获得ExternalClusterManager的具体实现类后，将调用ExternalClusterManager实现类的createTaskScheduler方法创建TaskScheduler。YarnClusterManager的createTaskScheduler方法的实现如下。

```
override def createTaskScheduler(sc: SparkContext, masterURL: String): TaskScheduler
    = {
  sc.deployMode match {
    case "cluster" => new YarnClusterScheduler(sc)
    case "client" => new YarnScheduler(sc)
    case _ => throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for
        Yarn")
  }
}
```



根据上述代码，如果SparkContext的deployMode属性是cluster，那么创建YarnClusterScheduler；如果SparkContext的deployMode属性是client，那么创建YarnScheduler。

根据代码清单4-18，在创建了TaskScheduler后，将调用ExternalClusterManager实现类的createSchedulerBackend方法创建SchedulerBackend。YarnClusterManager的createSchedulerBackend方法的实现如下。

```
override def createSchedulerBackend(sc: SparkContext,
    masterURL: String,
    scheduler: TaskScheduler): SchedulerBackend = {
  sc.deployMode match {
    case "cluster" =>
      new YarnClusterSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)
    case "client" =>
      new YarnClientSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerImpl], sc)
    case  _ =>
      throw new SparkException(s"Unknown deploy mode '${sc.deployMode}' for Yarn")
  }
}
```

根据上述代码，如果SparkContext的deployMode属性是cluster，那么创建YarnClusterSchedulerBackend；如果SparkContext的deployMode属性是client，那么创建YarnClientSchedulerBackend。

根据代码清单4-18，在创建完SchedulerBackend后，将调用ExternalClusterManager

实现类的 initialize 方法进行初始化。YarnClusterManager 的 initialize 方法的实现如下。

```
override def initialize(scheduler: TaskScheduler, backend: SchedulerBackend): Unit
    = {
  scheduler.asInstanceOf[TaskSchedulerImpl].initialize(backend)
}
```

根据上述代码，实际上调用了 TaskSchedulerImpl 的 initialize 方法。

根据 7.10.3 节的内容，我们知道在启动调度系统时会调用 YarnClusterSchedulerBackend 的 start 方法，其中主要确定了期望获得的 Executor 的数量，实现如下。

```
override def start() {
  val attemptId = ApplicationMaster.getAttemptId
  bindToYarn(attemptId.getApplicationId(), Some(attemptId))
  super.start()
  totalExpectedExecutors = YarnSparkHadoopUtil.getInitialTargetExecutorNumber(sc.
      conf)
}
```

Driver 初始化完毕会向 ApplicationMaster 进行注册，在 Yarn 部署模式中 Worker 已被 NodeManager 替代，ApplicationMaster 给 Application 分配资源主要借助于代码清单 9-96 中向 YARN 注册时返回的 allocator（类型为 YarnAllocator）。

关于 YARN 部署模式就简单介绍这些，更多内容可以访问 http://hadoop.apache.org/index.html 了解。

9.13.2 Mesos

Mesos 是诞生于 UC Berkeley 的另一个研究项目，现已成为 Apache Incubator 中的项目。Mesos 是一个集群管理器，提供了有效的、跨分布式应用或框架的资源隔离和共享，可以灵活支持 Hadoop、MPI、Hypertable、Spark 等。使用 ZooKeeper 实现容错复制，采用 Linux Container 对内存和 CPU 进行隔离。

Twitter 已经在使用 Mesos 管理集群资源。

Mesos 为了简化设计，采用 Master/Slave 架构，为了解决 Master 的单点故障，Master 非常轻量，仅保存了各种计算框架（如 Hadoop、MPI、Hypertable、Spark）和 Mesos slave 的一些状态，而这些状态很容易通过 framework 和 slave 重新注册而重构，因此很容易使用 ZooKeeper 解决 Mesos master 的单点故障问题。Mesos 的整体架构如图 9-35 所示。[⊖]

Mesos master 实际上是一个全局资源调度器，采用某种策略将某个 slave 上的空闲资源分配给各个计算框架，各种计算框架通过自己的调度器向 Mesos master 注册，以接入到 Mesos 中；而 Mesos slave 的主要功能是汇报任务的状态和启动各个计算框架的 executor（比如 Spark 的 Executor）。整个 Mesos 系统采用了双层调度框架：第一层，由 Mesos 将资源分配给框架；第二层，框架自己的调度器将资源分配给自己内部的任务。

⊖ 部分内容参考自博客 http://jiezhu2007.iteye.com/blog/2049073。

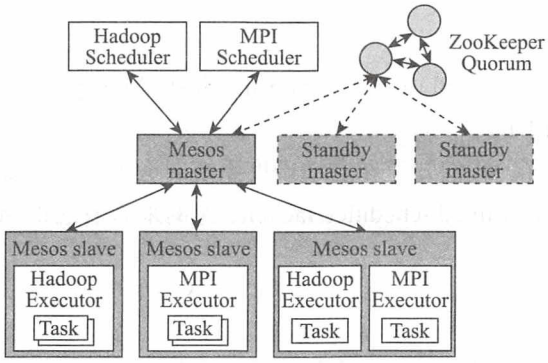


图 9-35 Mesos 的整体架构

在 Mesos 中，各种计算框架是完全融入 Mesos 中的，也就是说，如果你想在 Mesos 中添加一个新的计算框架，首先需要在 Mesos 中部署一套该框架。

如果我们设置 master 为 mesos://，那么在创建 TaskSchedulerImpl 时也只会匹配 masterUrl。由于 MesosClusterManager 的 canCreate 方法的实现如下。

```
override def canCreate(masterURL: String): Boolean = {
  masterURL.startsWith("mesos")
}
```

所以在调用 getClusterManager 方法后获得 MesosClusterManager。MesosClusterManager 的 createTaskScheduler 方法将创建 TaskSchedulerImpl，代码如下。

```
override def createTaskScheduler(sc: SparkContext, masterURL: String): TaskSche-
    duler = {
  new TaskSchedulerImpl(sc)
}
```

MesosClusterManager 的 createSchedulerBackend 方法将根据 spark.mesos.coarse 属性的值（默认为 true）决定是创建 MesosCoarseGrainedSchedulerBackend 还是 MesosFineGrained SchedulerBackend。MesosClusterManager 的 createSchedulerBackend 方法的实现如下。

```
override def createSchedulerBackend(sc: SparkContext,
    masterURL: String,
    scheduler: TaskScheduler): SchedulerBackend = {
  require(!sc.conf.get(IO_ENCRYPTION_ENABLED),
    "I/O encryption is currently not supported in Mesos.")

  val mesosUrl = MESOS_REGEX.findFirstMatchIn(masterURL).get.group(1)
  val coarse = sc.conf.getBoolean("spark.mesos.coarse", defaultValue = true)
  if (coarse) {
    new MesosCoarseGrainedSchedulerBackend(
      scheduler.asInstanceOf[TaskSchedulerImpl], sc, mesosUrl, sc.env.security-
          Manager)
  } else {
    new MesosFineGrainedSchedulerBackend(scheduler.asInstanceOf[TaskSchedulerIm
      pl], sc, mesosUrl)
```

```
    }
  }
```

MesosClusterManager 的 initialize 方法与 YarnClusterManager 的 initialize 方法的实现完全一样，这里就不再列出了。

根据 7.10.3 节的内容，我们知道在启动调度系统时会调用 SchedulerBackend 的 start 方法。此处以 MesosCoarseGrainedSchedulerBackend 为例来看看它的 start 方法的实现，代码如下。

```
override def start() {
  super.start()
  val driver = createSchedulerDriver( master, MesosCoarseGrainedSchedulerBackend.
     this,
    sc.sparkUser, sc.appName, sc.conf,
    sc.conf.getOption("spark.mesos.driver.webui.url").orElse(sc.ui.map(_.appUI-
        Address)),
    None, None, sc.conf.getOption("spark.mesos.driver.frameworkId")
  )

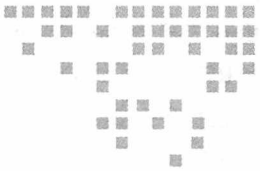
  unsetFrameworkID(sc)
  startScheduler(driver)
}
```

根据上述代码，MesosCoarseGrainedSchedulerBackend 的 start 方法通过调用 Mesos 的 API 完成 Driver 的注册。有关 Mesos 的 API 细节，请访问官网 http://mesos.apache.org/。

9.14 小结

我们最先了解 local 部署模式，然后精读 local-cluster 部署模式，最后分析 Standalone 部署模式。这样能够由简入难，由浅入深，尽可能帮助读者降低阅读 Spark 源码与理解部署模式架构原理的门槛。local 部署模式和 local-cluster 部署模式不能用于生成，为了简单，生成中可以选择使用 Standalone 部署模式。如果对 YARN 和 Mesos 也有一定的了解，那么使用 YARN 和 Mesos 也非常不错。

资源调度、资源回收及容错机制的源码剖析，除了能让我们理解 Spark 为什么能拥有很高的稳定性、可用性及容错能力外，其中各种架构设计的思想值得做架构设计的开发人员学习。



第 10 章 Chapter 10

Spark API

“我心素已闲，清川澹如此。请留盘石上，垂钓将已矣。”

——《清溪》

本章导读

王维的《清溪》是我喜欢的唐诗之一。每当我读到这首诗，王维那种淡泊名利、悠然自在的心境，也让我对田园、山水有了无限的向往！这首诗表面上写王维在黄花川游玩的快乐趣途，实际上也是对自己身心的修炼。只有心性崇高、返璞归真，才会使自己坦然、淡然、自然！

静下心来读读山水田园诗，徜徉在浪漫与理想中，固然是很多人的追求，可是如何将自己从繁忙的工作中解放出来，更为关键！ Spark 早期版本只提供了基于 SparkContext 的 API[⊖]。现在 Spark 又增加了 SparkSession 和 DataFrame 相关的 API，熟练使用它们，可以使大数据的开发工作变得更加简单。

word count 的例子是比较易于理解和讲解的大数据应用场景，所以 Hadoop 的入门例子选择了它。类似于任何介绍编程语言的 hello world 例子，word count 能用最短的时间，带给读者最直观的感受，从而降低学习曲线。作为大数据的入门实例，笔者为便于描述，自然选择经典的 word count 例子。

之所以讲解 Spark 的 word count 例子，也是为了能够把之前各章的内容串联起来。Spark

⊖ 有些对老版本的 Spark 熟悉的读者，还会提及 SQLContext、StreamingContext、GraphX、MLlib 等 API，但它们的底层实际都依赖于 SparkContext 的 API。

的任务运行在存储体系、调度系统及计算引擎之上。Spark Application、Application 的配置信息、Application 依赖的 Jar 包、map 任务的输出缓存、reduce 任务的缓存等都基于存储体系；程序转换为 RDD 构成的 DAG 后，RDD 的阶段划分、提交 Task、资源申请等都离不开调度系统；Task 在执行时对数据的缓存、聚合、Shuffle 管理等都由计算引擎支撑。

本章主要讲解的内容如下。

- DataType、Metadata、StructType 与 StructField 等基本概念。
- 数据源 DataSource。
- 检查点。
- RDD 的 API。
- 数据集合 Dataset 与 DataFrame。
- DataFrameReader。
- SparkSession。
- word count。

10.1 基本概念

本章会涉及一些基本概念，包括数据类型（DataType）、结构类型（StructType）、结构字段（StructField）及元数据（Metadata）。通过本节对这些基本概念的介绍，有利于本章后续内容的展开。

1. DataType 简介

DataType 是 Spark SQL 的所有数据类型的基本类型，Spark SQL 的所有数据类型都继承自 DataType，DataType 的继承体系如图 10-1 所示。

从图 10-1 中可以看到，Spark SQL 中定义的数据类型与 Java 的基本数据类型大部分都是一致的。由于 Spark SQL 不是本书要讲解的内容，所以读者在这里只需要了解这些内容即可。

2. Metadata 简介

Metadata 用来保存 StructField 的元数据信息，其本质是底层的 Map[String, Any]。Metadata 可以对 Boolean、Long、Double、String、Metadata、Array[Boolean]、Array[Long]、Array[Double]、Array[String] 和 Array[Metadata] 等类型的元数据进行存储或读取。Metadata 属于 Spark SQL 中的内容，这里不多介绍。

3. StructType 与 StructField

样例类 StructType 与样例类 StructField 共同构建起数据源的数据结构。

StructField 中共定义了 4 个属性：字段名称（name）、数据类型（dataType）、是否允许为 null（nullable）、元数据（metadata）。StructField 的定义如下。

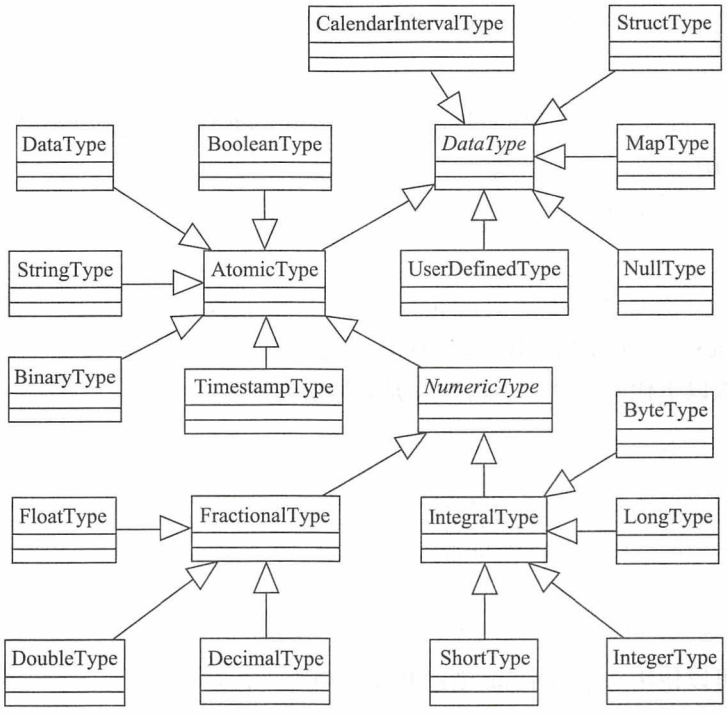


图 10-1 DataType 的继承体系

```
case class StructField(
  name: String,
  dataType: DataType,
  nullable: Boolean = true,
  metadata: Metadata = Metadata.empty)
```

StructType 的属性中最为重要的是类型为 Array[StructField] 的 fields，由此可以看出一个 StructType 中可以包括零到多个 StructField。为了便于理解，下面定义了一个简单的 StructType。

```
val struct =
   StructType(
     StructField("a", IntegerType, true) ::
     StructField("b", LongType, false) ::
     StructField("c", BooleanType, false) :: Nil)
```

对于构建的 struct，可以用以下语句获取名称为 b 的 StructField。

```
val singleField = struct("b")
```

如果要获取 struct 中不存在的 StructField，就像下面这样。

```
val nonExisting = struct("d")
```

那么 nonExisting 等于 null。

如果想要获取 struct 中的多个 StructField，需要像下面这样使用。

```
val twoFields = struct(Set("b", "c"))
```

此时的 twoFields 实际为 StructType(List(StructField(b,LongType,false), StructField(c,BooleanType,false)))。

如果像下面这样获取多个 StructField。

```
val ignoreNonExisting = struct(Set("b", "c", "d"))
```

那么其中的 d 会被忽略，此时的 ignoreNonExisting 实际为 StructType(List(StructField(b,LongType,false), StructField(c,BooleanType,false)))。

如果我们以以下代码定义一个新的数据结构 struct。

```
val innerStruct =
  StructType(
     StructField("f1", IntegerType, true) ::
     StructField("f2", LongType, false) ::
     StructField("f3", BooleanType, false) :: Nil)

 val struct = StructType(
   StructField("a", innerStruct, true) :: Nil)
```

那么我们可以使用这个 schema 创建 Row，就像下面这样。

```
val row = Row(Row(1, 2, true))
```

10.2 数据源 DataSource

从 Spark 1.3.0 开始，Spark 推出了 DataFrame 的 API，与此同时 DataSource 也被引入到 Spark 中。Spark 将文本文件、CSV 文件、JSON 文件等一系列格式的输入数据都作为数据源。特质 DataSourceRegister 是对所有数据源的抽象，DataSourceRegister 的所有具体实现都被注册到了 DataSource 中。DataSource 将负责对不同类型数据源的查找、创建不同类型数据源的信息、解析不同数据源的关系等。

10.2.1 DataSourceRegister 详解

DataSourceRegister 是数据源的抽象，所有数据源都应该实现它。DataSourceRegister 的定义非常简单，代码如下。

```
trait DataSourceRegister {
  def shortName(): String
}
```

shortName 方法意在获取数据源提供者使用的格式或格式的别名。

Spark 中实现了大量的数据源提供者，如图 10-2 所示。

这里以 TextFileFormat 实现的 shortName 方法为例。

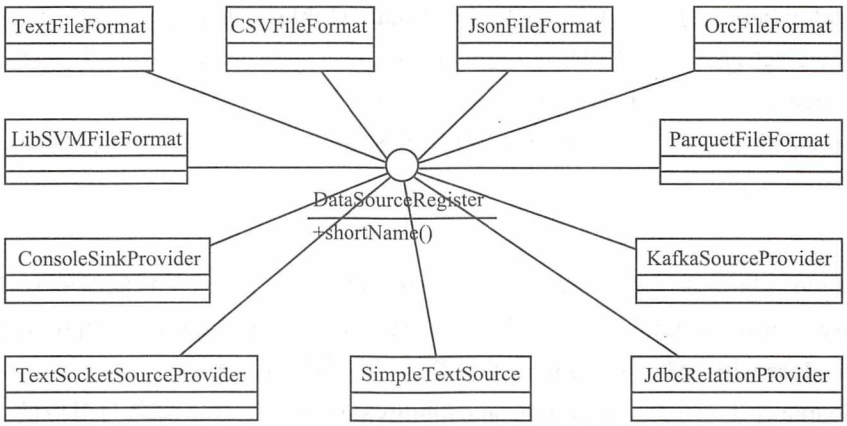


图 10-2　DataSourceRegister 的继承体系

```
override def shortName(): String = "text"
```

可以看到 TextFileFormat 的别名为 text。此外，其他 DataSourceRegister 的实现类也都有属于自己的别名，表 10-1 对此进行了展示。

表 10-1　DataSourceRegister 的子类及其别名

DataSourceRegister 的子类	别　名	DataSourceRegister 的子类	别　名
TextFileFormat	text	ConsoleSinkProvider	console
CSVFileFormat	csv	KafkaSourceProvider	kafka
JsonFileFormat	json	TextSocketSourceProvider	socket
OrcFileFormat	orc	JdbcRelationProvider	jdbc
LibSVMFileFormat	libsvm	SimpleTextSource	test
ParquetFileFormat	parquet		

10.2.2　DataSource 详解

DataSource 用于表示在 Spark SQL 中可插拔的数据源。DataSource 中包括以下属性。

- sparkSession：即 SparkSession。
- className：此名称用于决定要使用的类。
- paths：数据源的多个路径。
- userSpecifiedSchema：用户指定的 StructType。
- partitionColumns：分区字段的序列。
- bucketSpec ：类型为 BucketSpec。BucketSpec 是一种用于将数据集分解为更多可管理部分的技术。由于桶的数量是固定的，因此数据不会随数据而波动。
- options：类型为 Map[String, String]，用于保存选项。

- catalogTable：类型为 CatalogTable。CatalogTable 是用于定义表的字典。
- providingClass：数据源的类。providingClass 实际是以 className 为参数，调用 DataSource 伴生对象的 lookupDataSource 方法（见代码清单 10-1）获得的。
- sourceInfo：类型为 SourceInfo。样例类 SourceInfo 用于表示数据源的名称和元数据信息，其定义如下。

```
case class SourceInfo(name: String, schema: StructType, partitionColumns: Seq[String])
```

sourceInfo 实际是通过调用 DataSource 的 sourceSchema 方法（见代码清单 10-2）得到的。

- caseInsensitiveOptions：忽略大小写的选项配置。caseInsensitiveOptions 实际是通过 CaseInsensitiveMap 对 options 的包装，以忽略配置的大小写。

DataSource 伴生对象的 backwardCompatibilityMap 属性缓存了类名与对应的 DataSourceRegister 实现类之间的映射关系，代码如下。

```
private val backwardCompatibilityMap: Map[String, String] = {
  val jdbc = classOf[JdbcRelationProvider].getCanonicalName
  val json = classOf[JsonFileFormat].getCanonicalName
  val parquet = classOf[ParquetFileFormat].getCanonicalName
  val csv = classOf[CSVFileFormat].getCanonicalName
  val libsvm = "org.apache.spark.ml.source.libsvm.LibSVMFileFormat"
  val orc = "org.apache.spark.sql.hive.orc.OrcFileFormat"

  Map(
    "org.apache.spark.sql.jdbc" -> jdbc,
    "org.apache.spark.sql.jdbc.DefaultSource" -> jdbc,
    "org.apache.spark.sql.execution.datasources.jdbc.DefaultSource" -> jdbc,
    "org.apache.spark.sql.execution.datasources.jdbc" -> jdbc,
    "org.apache.spark.sql.json" -> json,
    "org.apache.spark.sql.json.DefaultSource" -> json,
    "org.apache.spark.sql.execution.datasources.json" -> json,
    "org.apache.spark.sql.execution.datasources.json.DefaultSource" -> json,
    "org.apache.spark.sql.parquet" -> parquet,
    "org.apache.spark.sql.parquet.DefaultSource" -> parquet,
    "org.apache.spark.sql.execution.datasources.parquet" -> parquet,
    "org.apache.spark.sql.execution.datasources.parquet.DefaultSource" -> parquet,
    "org.apache.spark.sql.hive.orc.DefaultSource" -> orc,
    "org.apache.spark.sql.hive.orc" -> orc,
    "org.apache.spark.ml.source.libsvm.DefaultSource" -> libsvm,
    "org.apache.spark.ml.source.libsvm" -> libsvm,
    "com.databricks.spark.csv" -> csv
  )
}
```

1. lookupDataSource

DataSource 的伴生对象中提供了根据指定的名字找到对应的数据源类的方法 lookupDataSource，其实现如代码清单 10-1 所示。

代码清单10-1 根据指定的名字查找数据源

```
def lookupDataSource(provider: String): Class[_] = {
```

```
    val provider1 = backwardCompatibilityMap.getOrElse(provider, provider)
    val provider2 = s"$provider1.DefaultSource"
    val loader = Utils.getContextOrSparkClassLoader
    val serviceLoader = ServiceLoader.load(classOf[DataSourceRegister], loader)

    try {
      serviceLoader.asScala.filter(_.shortName().equalsIgnoreCase(provider1)).toList
          match {
        case Nil =>
          try {
            Try(loader.loadClass(provider1)).orElse(Try(loader.loadClass(provider2)))
                match {
              case Success(dataSource) =>
                dataSource
              case Failure(error) =>
                // 忽略抛出异常的代码
            }
          } catch {
            // 忽略捕获异常的处理代码
          }
        case head :: Nil =>
          head.getClass
        case sources =>
          sys.error(s"Multiple sources found for $provider1 " +
            s"(${sources.map(_.getClass.getName).mkString(", ")}), " +
            "please specify the fully qualified class name.")
      }
    } catch {
        // 忽略捕获异常的处理代码
    }
  }
```

根据代码清单 10-1，lookupDataSource 方法的执行步骤如下。

1）从 backwardCompatibilityMap 中查找指定名称对应的数据源类 provider1，还在 provider1 后拼接 DefaultSource 构成 provider2。

2）由于所有的数据源类都必须实现 DataSourceRegister，且 DataSourceRegister 的 shortName 方法定义了获取数据源类的别名的规范，所以这里使用类加载器找到与 provider1 匹配的数据源类，如果未能匹配，则尝试加载由 provider1 或 provider2 指定的类。

DataSource 中提供了很多方法，限于篇幅不能一一介绍。笔者挑选出 DataSource 中与本章内容相关的一些方法进行介绍。

2. sourceSchema

sourceSchema 方法用于根据 providingClass 得到对应的 SourceInfo，其实现如代码清单 10-2 所示。

代码清单10-2 根据providingClass 构造SourceInfo

```
  private def sourceSchema(): SourceInfo = {
    providingClass.newInstance() match {
      case s: StreamSourceProvider =>
```

```
        val (name, schema) = s.sourceSchema(
          sparkSession.sqlContext, userSpecifiedSchema, className, caseInsensitive-
              Options)
        SourceInfo(name, schema, Nil)

      case format: FileFormat =>
        val path = caseInsensitiveOptions.getOrElse("path", {
          throw new IllegalArgumentException("'path' is not specified")
        })

        val hdfsPath = new Path(path)
        if (!SparkHadoopUtil.get.isGlobPath(hdfsPath)) {
          val fs = hdfsPath.getFileSystem(sparkSession.sessionState.newHadoop-
              Conf())
          if (!fs.exists(hdfsPath)) {
            throw new AnalysisException(s"Path does not exist: $path")
          }
        }

        val isSchemaInferenceEnabled = sparkSession.sessionState.conf.streaming-
            SchemaInference
        val isTextSource = providingClass == classOf[text.TextFileFormat]
        // 省略抛出IllegalArgumentException的代码
        val (dataSchema, partitionSchema) = getOrInferFileFormatSchema(format)
        SourceInfo(
          s"FileSource[$path]",
          StructType(dataSchema ++ partitionSchema),
          partitionSchema.fieldNames)

      case _ =>
        throw new UnsupportedOperationException(
          s"Data source $className does not support streamed reading")
    }
  }
```

根据代码清单 10-2，sourceSchema 方法通过反射得到 providingClass 对应的实例，然后根据实例的不同类型构造 SourceInfo。

3. resolveRelation

resolveRelation 方法用于根据数据源的提供类（providingClass）和 userSpecifiedSchema 创建读写数据源所需的 BaseRelation。由于 resolveRelation 方法的内容很多，这里只展示 resolveRelation 方法的一部分内容，如代码清单 10-3 所示。

代码清单10-3 resolveRelation方法的部分实现

```
def resolveRelation(checkFilesExist: Boolean = true): BaseRelation = {
  val relation = (providingClass.newInstance(), userSpecifiedSchema) match {
    case (dataSource: SchemaRelationProvider, Some(schema)) =>
      dataSource.createRelation(sparkSession.sqlContext, caseInsensitiveOptions,
          schema)
    case (dataSource: RelationProvider, None) =>
      dataSource.createRelation(sparkSession.sqlContext, caseInsensitiveOptions)
    case (_: SchemaRelationProvider, None) =>
```

```
        // 忽略创建BaseRelation的代码
      case (format: FileFormat, _) =>
        // 忽略创建BaseRelation的代码
      case _ =>
        throw new AnalysisException(
          s"$className is not a valid Spark SQL Data Source.")
    }
    relation
  }
```

10.3 检查点的实现

检查点是很多分布式系统为了容灾容错引入的机制，其实质是将系统运行期的内存数据结构和状态持久化到磁盘上，在需要时通过对这些持久化数据的读取，重新构造出之前的运行期状态。Spark 使用检查点主要是为了将 RDD 的执行状态保留下来，在重新执行时就不用重新计算，而直接从检查点读取。CheckpointRDD 是对检查点数据进行操作的 RDD，例如，读写检查点数据。RDDCheckpointData 表示 RDD 检查点的数据。Spark 的检查点离不开 CheckpointRDD 和 RDDCheckpointData 的支持，本节将对它们的代码实现进行分析。

10.3.1 CheckpointRDD 的实现

CheckpointRDD 是特殊的 RDD，用来从存储体系中恢复检查点数据。CheckpointRDD 的定义如下。

```
private[spark] abstract class CheckpointRDD[T: ClassTag](sc: SparkContext)
extends RDD[T](sc, Nil) {

override def doCheckpoint(): Unit = { }
override def checkpoint(): Unit = { }
override def localCheckpoint(): this.type = this

protected override def getPartitions: Array[Partition] = ???
override def compute(p: Partition, tc: TaskContext): Iterator[T] = ???
}
```

可以看到 CheckpointRDD 重写了 RDD 的 5 个方法，分别如下。

- doCheckpoint：保存检查点。
- checkpoint：读取检查点数据。
- localCheckpoint：本地的检查点。
- getPartitions：获取检查点的分区数组。
- compute：名为计算，实际是从检查点恢复数据。

CheckpointRDD 有 LocalCheckpointRDD 和 ReliableCheckpointRDD 两个子类，它们都没有完全实现 CheckpointRDD 中的方法。下面以 ReliableCheckpointRDD 为例，来介绍 Check-

pointRDD 的具体实现。

ReliableCheckpointRDD 中定义了以下属性。

- sc：即 SparkContext。
- checkpointPath：检查点目录的字符串表示。
- _partitioner: 调用方指定的分区计算器，默认为 None。
- hadoopConf：SparkContext 的 _hadoopConfiguration 属性，即 Hadoop 的配置信息。
- cpath：类型为 org.apache.hadoop.fs.Path，表示 checkpointPath 对应的 Hadoop 文件系统中的路径。
- fs：使用 hadoopConf 得到的 org.apache.hadoop.fs.FileSystem。
- broadcastedConf：调用 SparkContext 的 broadcast 方法对 hadoopConf 进行广播后返回的 Broadcast 对象。
- partitioner：ReliableCheckpointRDD 的分区计算器，优先采用 _partitioner 指定的，否则调用 ReliableCheckpointRDD 的伴生对象的 readCheckpointedPartitionerFile 方法从 checkpointPath 指定的检查点目录下读取分区计算器。

ReliableCheckpointRDD 的伴生对象中提供了很多工具方法，下面逐个介绍。

1. writePartitionToCheckpointFile

writePartitionToCheckpointFile 方法用于将 RDD 分区的数据写入到检查点目录下的文件中，其实现如代码清单 10-4 所示。

代码清单10-4 writePartitionToCheckpointFile方法

```
def writePartitionToCheckpointFile[T: ClassTag](
    path: String,
    broadcastedConf: Broadcast[SerializableConfiguration],
    blockSize: Int = -1)(ctx: TaskContext, iterator: Iterator[T]) {
  val env = SparkEnv.get
  val outputDir = new Path(path)
  val fs = outputDir.getFileSystem(broadcastedConf.value.value)

  val finalOutputName = ReliableCheckpointRDD.checkpointFileName(ctx.partition-
    Id())
  val finalOutputPath = new Path(outputDir, finalOutputName)
  val tempOutputPath =
    new Path(outputDir, s".$finalOutputName-attempt-${ctx.attemptNumber()}")

  val bufferSize = env.conf.getInt("spark.buffer.size", 65536)

  val fileOutputStream = if (blockSize < 0) {
    fs.create(tempOutputPath, false, bufferSize)
  } else {
    // 省略与测试相关的代码
  }
  val serializer = env.serializer.newInstance()
  val serializeStream = serializer.serializeStream(fileOutputStream)
  Utils.tryWithSafeFinally {
```

```
      serializeStream.writeAll(iterator)
    }   {
      serializeStream.close()
    }

    if (!fs.rename(tempOutputPath, finalOutputPath)) {
      if (!fs.exists(finalOutputPath)) {
        logInfo(s"Deleting tempOutputPath $tempOutputPath")
        fs.delete(tempOutputPath, false)
        throw new IOException("Checkpoint failed: failed to save output of task: " +
          s"${ctx.attemptNumber()} and final output path does not exist: $finalOut-
              putPath")
      } else {
        // Some other copy of this task must've finished before us and renamed it
        logInfo(s"Final output path $finalOutputPath already exists; not overwrit-
            ing it")
        if (!fs.delete(tempOutputPath, false)) {
          logWarning(s"Error deleting ${tempOutputPath}")
        }
      }
    }
  }
```

2. writePartitionerToCheckpointDir

writePartitionerToCheckpointDir 方法用于将分区计算器的数据写入到检查点的目录下，其实现如代码清单 10-5 所示。

代码清单10-5 writePartitionerToCheckpointDir方法

```
  private def writePartitionerToCheckpointDir(
    sc: SparkContext, partitioner: Partitioner, checkpointDirPath: Path): Unit = {
    try {
      val partitionerFilePath = new Path(checkpointDirPath, checkpointPartitioner-
          FileName)
      val bufferSize = sc.conf.getInt("spark.buffer.size", 65536)
      val fs = partitionerFilePath.getFileSystem(sc.hadoopConfiguration)
      val fileOutputStream = fs.create(partitionerFilePath, false, bufferSize)
      val serializer = SparkEnv.get.serializer.newInstance()
      val serializeStream = serializer.serializeStream(fileOutputStream)
      Utils.tryWithSafeFinally {
        serializeStream.writeObject(partitioner)
      } {
        serializeStream.close()
      }
      logDebug(s"Written partitioner to $partitionerFilePath")
    } catch {
      case NonFatal(e) =>
        logWarning(s"Error writing partitioner $partitioner to $checkpointDirPath")
    }
  }
```

3. writeRDDToCheckpointDirectory

writeRDDToCheckpointDirectory 方法用于将 RDD 的数据写入检查点目录，其实现如代

码清单 10-6 所示。

代码清单10-6 writeRDDToCheckpointDirectory方法

```
def writeRDDToCheckpointDirectory[T: ClassTag](
    originalRDD: RDD[T],
    checkpointDir: String,
    blockSize: Int = -1): ReliableCheckpointRDD[T] = {

  val sc = originalRDD.sparkContext

  val checkpointDirPath = new Path(checkpointDir)
  val fs = checkpointDirPath.getFileSystem(sc.hadoopConfiguration)
  if (!fs.mkdirs(checkpointDirPath)) {
    throw new SparkException(s"Failed to create checkpoint path $checkpoint-
        DirPath")
  }

  val broadcastedConf = sc.broadcast(
    new SerializableConfiguration(sc.hadoopConfiguration))
  sc.runJob(originalRDD,
    writePartitionToCheckpointFile[T](checkpointDirPath.toString, broadcastedConf) _)

  if (originalRDD.partitioner.nonEmpty) {
    writePartitionerToCheckpointDir(sc, originalRDD.partitioner.get, checkpointDirPath)
  }

  val newRDD = new ReliableCheckpointRDD[T](
    sc, checkpointDirPath.toString, originalRDD.partitioner)
  if (newRDD.partitions.length != originalRDD.partitions.length) {
    throw new SparkException(
      s"Checkpoint RDD $newRDD(${newRDD.partitions.length}) has different " +
        s"number of partitions from original RDD $originalRDD(${originalRDD.
            partitions.length})")
  }
  newRDD
}
```

根据代码清单 10-6，writeRDDToCheckpointDirectory 方法的执行步骤如下。

1）调用 SparkContext 的 runJob 方法将 RDD 的数据写入到检查点目录。将数据写入磁盘的函数是 ReliableCheckpointRDD 的伴生对象的 writePartitionToCheckpointFile 方法，其实现如代码清单 10-4 所示。

2）如果 RDD 有分区计算器，那么调用 ReliableCheckpointRDD 的伴生对象的 writePartitionerToCheckpointDir 方法（见代码清单 10-5）将分区计算器的信息也写入到检查点目录。

3）创建并返回 ReliableCheckpointRDD。

4. readCheckpointFile

readCheckpointFile 方法用于从检查点目录下的文件中读取 RDD 的数据，其实现如代码清单 10-7 所示。

代码清单10-7 readCheckpointFile方法

```
def readCheckpointFile[T](
    path: Path,
    broadcastedConf: Broadcast[SerializableConfiguration],
    context: TaskContext): Iterator[T] = {
  val env = SparkEnv.get
  val fs = path.getFileSystem(broadcastedConf.value.value)
  val bufferSize = env.conf.getInt("spark.buffer.size", 65536)
  val fileInputStream = fs.open(path, bufferSize)
  val serializer = env.serializer.newInstance()
  val deserializeStream = serializer.deserializeStream(fileInputStream)

  context.addTaskCompletionListener(context => deserializeStream.close())

  deserializeStream.asIterator.asInstanceOf[Iterator[T]]
}
```

上面介绍了 ReliableCheckpointRDD 的伴生对象提供的方法，下面介绍 ReliableCheckpointRDD 对父类 RDD 重写的部分方法。

5. getPartitions

ReliableCheckpointRDD 实现的 getPartitions 方法（见代码清单 10-8）用于从检查点文件中获取分区数组。

代码清单10-8 getPartitions方法

```
protected override def getPartitions: Array[Partition] = {
  val inputFiles = fs.listStatus(cpath)
    .map(_.getPath)
    .filter(_.getName.startsWith("part-"))
    .sortBy(_.getName.stripPrefix("part-").toInt)
  inputFiles.zipWithIndex.foreach { case (path, i) =>
    if (path.getName != ReliableCheckpointRDD.checkpointFileName(i)) {
      throw new SparkException(s"Invalid checkpoint file: $path")
    }
  }
  Array.tabulate(inputFiles.length)(i => new CheckpointRDDPartition(i))
}
```

6. getPreferredLocations

ReliableCheckpointRDD 实现的 getPreferredLocations 方法（见代码清单 10-9）用于从检查点文件中获取偏好位置。

代码清单10-9 getPreferredLocations方法

```
protected override def getPreferredLocations(split: Partition): Seq[String] = {
  val status = fs.getFileStatus(
    new Path(checkpointPath, ReliableCheckpointRDD.checkpointFileName(split.index)))
  val locations = fs.getFileBlockLocations(status, 0, status.getLen)
  locations.headOption.toList.flatMap(_.getHosts).filter(_ != "localhost")
}
```

7. compute

ReliableCheckpointRDD 实现的 compute 方法（见代码清单 10-10）用于从检查点文件中获取检查点数据。

代码清单10-10 compute方法

```
override def compute(split: Partition, context: TaskContext): Iterator[T] = {
  val file = new Path(checkpointPath, ReliableCheckpointRDD.checkpointFile-
     Name(split.index))
  ReliableCheckpointRDD.readCheckpointFile(file, broadcastedConf, context)
}
```

根据对 CheckpointRDD 的各个方法的分析，检查点似乎不再那么神秘，其实现不过是对文件的读与写。

10.3.2 RDDCheckpointData 的实现

RDDCheckpointData 用于保存与检查点相关的信息。每个 RDDCheckpointData 实例都与一个 RDD 实例相关联。RDDCheckpointData 中一共有三个属性。

- rdd：RDDCheckpointData 关联的 RDD。
- cpState：检查点的状态。默认为 Initialized。cpState 的值来自枚举类型 CheckpointState，CheckpointState 中定义了检查点的状态，包括初始化完成（Initialized）、正在保存检查点（CheckpointingInProgress）和保存检查点完毕（Checkpointed）。
- cpRDD：保存检查点数据的 RDD，即 CheckpointRDD 的实现类。

RDDCheckpointData 中定义了一些方法，分别如下。

1. isCheckpointed

isCheckpointed 方法用于判断是否已经为 RDDCheckpointData 关联的 RDD 保存了检查点数据，其实现如下。

```
def isCheckpointed: Boolean = RDDCheckpointData.synchronized {
  cpState == Checkpointed
}
```

2. checkpoint

checkpoint 方法是用于将 RDDCheckpointData 关联的 RDD 的数据保存到检查点的模板方法，其实现如代码清单 10-11 所示。

代码清单10-11 保存检查点的模板方法checkpoint

```
final def checkpoint(): Unit = {
  RDDCheckpointData.synchronized {
    if (cpState == Initialized) {
      cpState = CheckpointingInProgress
    } else {
      return
```

```
      }
    }
    val newRDD = doCheckpoint()

    // Update our state and truncate the RDD lineage
    RDDCheckpointData.synchronized {
      cpRDD = Some(newRDD)
      cpState = Checkpointed
      rdd.markCheckpointed()
    }
  }
```

根据代码清单 10-11，checkpoint 方法的执行步骤如下。

1）如果检查点的状态是 Initialized，那么将 cpState 设置为 CheckpointingInProgress，否则返回。

2）调用 doCheckpoint 方法保存检查点并生成 CheckpointRDD。doCheckpoint 方法需要 RDDCheckpointData 的子类实现。RDDCheckpointData 的子类有 LocalRDDCheckpointData 和 ReliableRDDCheckpointData 两种，10.3.3 节以 ReliableRDDCheckpointData 为例，介绍了其实现的 doCheckpoint 方法。

3）由 cpRDD 持有刚生成的 CheckpointRDD，然后将 cpState 设置为 Checkpointed，最后调用 RDD 的 markCheckpointed 方法（见代码清单 10-12）清空依赖。之所以清空依赖，是因为现在已经有了 CheckpointRDD，之前的依赖关系不再需要了。

代码清单10-12 标记RDD已经保存了检查点

```
private[spark] def markCheckpointed(): Unit = {
  clearDependencies()
  partitions_ = null
  deps = null    // Forget the constructor argument for dependencies too
}

protected def clearDependencies() {
  dependencies_ = null
}
```

3. checkpointRDD

checkpointRDD 方法用于获取 cpRDD 持有的 CheckpointRDD，其实现如下。

```
def checkpointRDD: Option[CheckpointRDD[T]] = RDDCheckpointData.synchronized { cpRDD }
```

4. getPartitions

getPartitions 方法用于获取 CheckpointRDD 的分区数组，其实现如下。

```
def getPartitions: Array[Partition] = RDDCheckpointData.synchronized {
  cpRDD.map(_.partitions).getOrElse { Array.empty }
}
```

有了对 RDDCheckpointData 的理解，下一小节以 RDDCheckpointData 的子类 ReliableRDDCheckpointData 为例，来看看 RDDCheckpointData 该如何实现。

10.3.3 ReliableRDDCheckpointData 的实现

本节以 RDDCheckpointData 的子类 ReliableRDDCheckpointData 为例，来看看 RDDCheckpointData 的具体实现。

ReliableRDDCheckpointData 除继承了 RDDCheckpointData 的属性外，还有自身的一个属性 cpDir。cpDir 是保存 ReliableRDDCheckpointData 所关联的 RDD 数据的检查点目录，是通过调用 ReliableRDDCheckpointData 的伴生对象的 checkpointPath 方法（见代码清单 10-13）生成的。

ReliableRDDCheckpointData 及其伴生对象提供了以下方法。

1. checkpointPath

checkpointPath 方法用于在 SparkContext 的 checkpointDir 属性指定的 RDD 计算过程中保存检查点的目录下创建子目录，作为保存 ReliableRDDCheckpointData 所关联的 RDD 数据的检查点目录。ReliableRDDCheckpointData 的伴生对象提供的 checkpointPath 方法如代码清单 10-13 所示。

代码清单10-13 创建保存RDD数据的检查点目录

```
def checkpointPath(sc: SparkContext, rddId: Int): Option[Path] = {
  sc.checkpointDir.map { dir => new Path(dir, s"rdd-$rddId") }
}
```

2. doCheckpoint

ReliableRDDCheckpointData 实现了父类 RDDCheckpointData 定义的 doCheckpoint 方法如代码清单 10-14 所示。

代码清单10-14 保存检查点

```
protected override def doCheckpoint(): CheckpointRDD[T] = {
  val newRDD = ReliableCheckpointRDD.writeRDDToCheckpointDirectory(rdd, cpDir)

  if (rdd.conf.getBoolean("spark.cleaner.referenceTracking.cleanCheckpoints",
      false)) {
    rdd.context.cleaner.foreach { cleaner =>
      cleaner.registerRDDCheckpointDataForCleanup(newRDD, rdd.id)
    }
  }

  logInfo(s"Done checkpointing RDD ${rdd.id} to $cpDir, new parent is RDD
      ${newRDD.id}")
  newRDD
}
```

根据代码清单 10-14，ReliableRDDCheckpointData 实现的 doCheckpoint 方法的执行步骤如下。

1）调用 ReliableCheckpointRDD 的伴生对象的 writeRDDToCheckpointDirectory 方法

（见代码清单 10-6）将 RDD 的数据保存到检查点目录。

2）如果 spark.cleaner.referenceTracking.cleanCheckpoints 属性指定为 true，那么将生成的 ReliableCheckpointRDD 注册到 SparkContext 的子组件 ContextCleaner 的 referenceBuffer 中，以便于 ContextCleaner 对 ReliableCheckpointRDD 进行清理。

3）返回生成的 ReliableCheckpointRDD。

10.4　RDD 的再次分析

笔者早在 7.2.2 节就对 RDD 的实现进行了分析，但当时只介绍了与调度系统相关的 API。RDD 还提供了很多其他类型的 API，包括对 RDD 进行转换的 API、对 RDD 进行计算（动作）的 API 及 RDD 检查点相关的 API。转换 API 里的计算是延迟的，也就是说调用转换 API 不会向 Spark 集群提交 Job，更不会执行转换计算。只有调用了动作 API，才会提交 Job 并触发对转换计算的执行。由于 RDD 提供的 API 非常多，本书不可能一一展示。由于在 10.8 节将要介绍 word count 的例子，因此本节主要挑选 word count 例子中使用到的 API 进行分析。

10.4.1　转换 API

转换（transform）是指对现有 RDD 执行某个函数后转换为新的 RDD 的过程。转换前的 RDD 与转换后的 RDD 之间具有依赖和血缘关系。RDD 的多次转换将创建出多个 RDD，这些 RDD 构成了一张单向依赖的图，也就是 DAG。下面挑选 10.8 节的 word count 例子所涉及的转换 API 进行介绍。

1. mapPartitions

mapPartitions 方法用于将 RDD 转换为 MapPartitionsRDD，其实现如代码清单 10-15 所示。

代码清单10-15　RDD的mapPartitions方法

```
def mapPartitions[U: ClassTag](
    f: Iterator[T] => Iterator[U],
    preservesPartitioning: Boolean = false): RDD[U] = withScope {
  val cleanedF = sc.clean(f)
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(iter),
    preservesPartitioning)
}
```

为便于理解，这里假设函数 f 的作用是过滤出大于 0 的数字，那么 mapPartitions 方法的执行可以用图 10-3 表示。

仅供非商业用途或交流学习使用

2. mapPartitionsWithIndex

mapPartitionsWithIndex 方法（见代码清单 10-16）用于创建一个将与分区索引相关的函数应用到 RDD 的每个分区的 MapPartitionsRDD。

代码清单10-16 RDD的mapPartitionsWithIndex方法

```
def mapPartitionsWithIndex[U: ClassTag](
    f: (Int, Iterator[T]) => Iterator[U],
    preservesPartitioning: Boolean = false): RDD[U] = withScope {
  val cleanedF = sc.clean(f)
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => cleanedF(index, iter),
    preservesPartitioning)
}
```

mapPartitionsWithIndex 与 mapPartitions 相似，区别在于多接收分区索引的参数。我们假设函数 f 的作用是将每个分区的数字累加并且与分区索引以逗号分隔输出，那么 mapPartitionsWithIndex 方法的执行可以用图 10-4 表示。

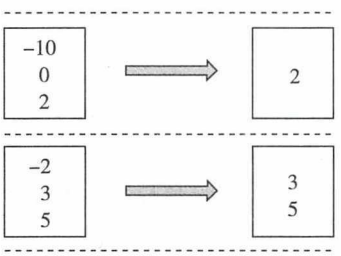


图 10-3 mapPartitions 示例

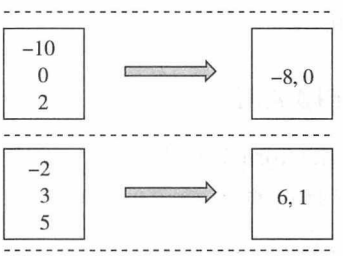


图 10-4 mapPartitionsWithIndex 示例

3. mapPartitionsWithIndexInternal

mapPartitionsWithIndexInternal 方法（见代码清单 10-17）用于创建一个将函数应用到 RDD 的每个分区的 MapPartitionsRDD。由于此方法是私有的，所以只在 Spark SQL 内部使用。

代码清单10-17 RDD的mapPartitionsWithIndexInternal方法

```
private[spark] def mapPartitionsWithIndexInternal[U: ClassTag](
    f: (Int, Iterator[T]) => Iterator[U],
    preservesPartitioning: Boolean = false): RDD[U] = withScope {
  new MapPartitionsRDD(
    this,
    (context: TaskContext, index: Int, iter: Iterator[T]) => f(index, iter),
    preservesPartitioning)
}
```

4. flatMap

flatMap 方法（见代码清单 10-18）用于向 RDD 中的所有元素应用函数，并对结果扁平化处理。

仅供非商业用途或交流学习使用

代码清单10-18 RDD的flatMap方法

```
def flatMap[U: ClassTag](f: T => TraversableOnce[U]): RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.flatMap(cleanF))
}
```

根据代码清单 10-18，flatMap 方法也将返回 MapPartitionsRDD。我们假设函数 f 的作用是将给每个数字加上 5，那么 flatMap 方法的执行可以用图 10-5 表示。

5. map

map 方法（见代码清单 10-19）用于向 RDD 中的所有元素应用函数。

代码清单10-19 RDD的map方法

```
def map[U: ClassTag](f: T => U): RDD[U] = withScope {
  val cleanF = sc.clean(f)
  new MapPartitionsRDD[U, T](this, (context, pid, iter) => iter.map(cleanF))
}
```

我们假设函数 f 的作用是将给每个数字加上 5，那么 map 方法的执行可以用图 10-6 表示。

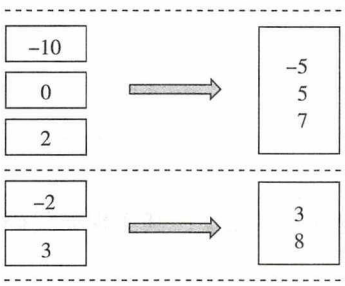


图 10-5 flatMap 示例

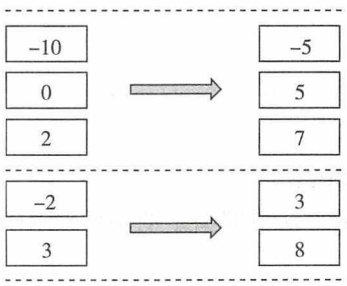


图 10-6 map 示例

6. toJavaRDD

toJavaRDD 方法用于将 RDD 自己转换为 JavaRDD，其实现如代码清单 10-20 所示。

代码清单10-20 RDD的toJavaRDD方法

```
def toJavaRDD() : JavaRDD[T] = {
  new JavaRDD(this)(elementClassTag)
}
```

有了以上几个转换 API 的分析，读者对于转换的含义应该有更清楚的认识了。

10.4.2 动作 API

由于转换 API 都是预先编织好，但是不会执行的，所以 Spark 需要一些 API 来触发对转换的执行。动作 API 触发对数据的转换后，将接收到一些结果数据，动作 API 因此还具

仅供非商业用途或交流学习使用

备对这些数据进行收集、遍历、叠加的功能。下面挑选 10.8 节的 word count 例子使用的动作 API——collect 进行介绍。此外再介绍 foreach 和 reduce 两个动作 API。

1. collect

collect 方法（见代码清单 10-21）将调用 SparkContext 的 runJob 方法（见代码清单 4-29）提交基于 RDD 的所有分区上的作业，并返回数组形式的结果。

代码清单10-21 RDD的collect方法

```
def collect(): Array[T] = withScope {
  val results = sc.runJob(this, (iter: Iterator[T]) => iter.toArray)
  Array.concat(results: _*)
}
```

collect 可以用图 10-7 来表示。

2. foreach

foreach 方法（见代码清单 10-22）将调用 SparkContext 的 runJob 方法（见代码清单 4-29）提交将函数应用到 RDD 中所有元素的作业。

代码清单10-22 RDD的foreach方法

```
def foreach(f: T => Unit): Unit = withScope {
  val cleanF = sc.clean(f)
  sc.runJob(this, (iter: Iterator[T]) => iter.foreach(cleanF))
}
```



3. reduce

reduce 方法按照指定的函数对 RDD 中的元素进行叠加操作，其实现如代码清单 10-23 所示。

代码清单10-23 RDD的reduce方法

```
def reduce(f: (T, T) => T): T = withScope {
  val cleanF = sc.clean(f)
  val reducePartition: Iterator[T] => Option[T] = iter => {
    if (iter.hasNext) {
      Some(iter.reduceLeft(cleanF))
    } else {
      None
    }
  }
  var jobResult: Option[T] = None
  val mergeResult = (index: Int, taskResult: Option[T]) => {
    if (taskResult.isDefined) {
      jobResult = jobResult match {
        case Some(value) => Some(f(value, taskResult.get))
        case None => taskResult
      }
    }
```

仅供非商业用途或交流学习使用


```
    }
    sc.runJob(this, reducePartition, mergeResult)
    jobResult.getOrElse(throw new UnsupportedOperationException("empty collection"))
  }
```

为了便于说明 reduce 的作用，这里假设函数 f 的定义是：(L,R)=>(L+R)，那么可以用图 10-8 来表示 reduce 的效果。

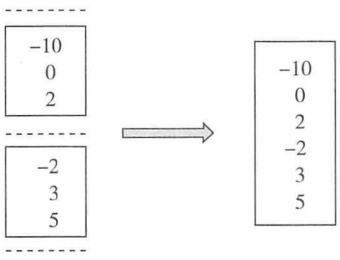


图 10-7 collect 的示例

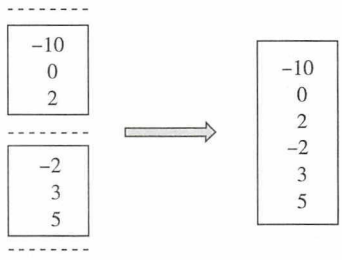


图 10-8 reduce 示例

10.4.3 检查点 API 的实现分析

RDD 中提供了很多与检查点相关的 API，通过对这些 API 的使用，Spark 应用程序才能够启用、保存及使用检查点，提高应用程序的容灾和容错能力。下面进行介绍。



1. 检查点的启用

用户提交的 Spark 作业必须主动调用 RDD 的 checkpoint 方法（见代码清单 10-24），才会启动检查点功能。

代码清单10-24 RDD的checkpoint方法

```
def checkpoint(): Unit = RDDCheckpointData.synchronized {
  if (context.checkpointDir.isEmpty) {
      throw new SparkException("Checkpoint directory has not been set in the
SparkContext")
  } else if (checkpointData.isEmpty) {
    checkpointData = Some(new ReliableRDDCheckpointData(this))
  }
}
```

根据代码清单 10-24，给 SparkContext 指定 checkpointDir 是启用检查点机制的前提。可以使用 SparkContext 的 setCheckpointDir 方法（见代码清单 4-30）设置 checkpointDir。如果没有指定 RDDCheckpointData，那么创建 ReliableRDDCheckpointData。

2. 检查点的保存

RDD 的 doCheckpoint 方法用于将 RDD 的数据保存到检查点，其实现如代码清单 10-25 所示。由于此方法是私有的，只能在 RDD 内部使用。

仅供非商业用途或交流学习使用

代码清单10-25 RDD的doCheckpoint方法

```
private[spark] def doCheckpoint(): Unit = {
  RDDOperationScope.withScope(sc, "checkpoint", allowNesting = false, ignore-
      Parent = true) {
    if (!doCheckpointCalled) {
      doCheckpointCalled = true
      if (checkpointData.isDefined) {
        if (checkpointAllMarkedAncestors) {
          dependencies.foreach(_.rdd.doCheckpoint())
        }
        checkpointData.get.checkpoint()
      } else {
        dependencies.foreach(_.rdd.doCheckpoint())
      }
    }
  }
}
```

根据代码清单 10-25，doCheckpoint 方法的执行步骤如下。

1）如果 checkpointData 中保存了 RDDCheckpointData，调用 RDDCheckpointData 的 checkpoint 方法（见代码清单 10-11）保存检查点。如果需要对祖先 RDD 保存检查点，那么还会调用每个依赖的 RDD 的 doCheckpoint 方法。由于在启用检查点时，保存到 checkpointData 中的是 RDDCheckpointData 的子类 ReliableRDDCheckpointData，因此 RDDCheckpointData 的 checkpoint 方法中将调用 ReliableRDDCheckpointData 的 doCheckpoint 方法（见代码清单 10-14）。

2）如果 checkpointData 中没有保存 RDDCheckpointData，那么调用每个依赖的 RDD 的 doCheckpoint 方法。

3. 检查点的使用

在 7.2.2 节曾介绍过获取 RDD 的分区数组的 partitions 方法（见代码清单 7-1）、获取指定分区的偏好位置的 preferredLocations 方法（见代码清单 7-2）、获取当前 RDD 的所有依赖的 dependencies 方法（见代码清单 7-3）。虽然这几个方法的作用不同，但是实现方式却是类似的，即首先从 RDD 关联的 CheckpointRDD 中查找对应信息。

根据对检查点的启用和保存的分析，负责为 RDD 提供检查点服务的实际是 ReliableCheckpointRDD。因此当调用 RDD 的 partitions 方法时，会首先调用 ReliableCheckpointRDD 的 partitions 方法，进而调用 ReliableCheckpointRDD 的 getPartitions 方法（见代码清单 10-8），最后才调用 RDD 自己的 getPartitions 方法。当调用 RDD 的 preferredLocations 方法时，首先会调用 ReliableCheckpointRDD 的 getPreferredLocations 方法（见代码清单 10-9），当调用 RDD 的 dependencies 方法时，首先会尝试将 ReliableCheckpointRDD 封装为 OneToOneDependency。

除了以上场景外，对 RDD 的迭代计算也涉及对检查点的使用，其中将调用 ReliableCheckpointRDD 的 compute 方法（见代码清单 10-10）。迭代计算的内容将在下一小节介绍。

仅供非商业用途或交流学习使用

10.4.4 迭代计算

在 8.5.3 节和 8.5.4 节分析 ShuffleMapTask 和 ResultTask 的 runTask 方法时已经看到，Task 的执行离不开对 RDD 的 iterator 方法的调用。RDD 的 iterator 方法是迭代计算的入口，其实现如代码清单 10-26 所示。

代码清单10-26 迭代计算的入口

```
final def iterator(split: Partition, context: TaskContext): Iterator[T] = {
  if (storageLevel != StorageLevel.NONE) {
    getOrCompute(split, context)
  } else {
    computeOrReadCheckpoint(split, context)
  }
}
```

根据代码清单 10-26，iterator 方法的执行步骤如下。

1）如果 RDD 的存储级别（StorageLevel）不是 NONE，那么根据 6.1.2 节对 StorageLevel 的分析，StorageLevel 的构造器是私有的，而 Spark 内置的存储级别都定义在代码清单 6-7 中。这些内置的存储级别除 NONE 外，至少会使用磁盘、堆内内存、堆外内存三者之一，因此可以调用 getOrCompute 方法（见代码清单 10-27）从这些存储中尝试获取计算结果。

2）如果 RDD 的存储级别（StorageLevel）是 NONE，那么说明分区任务可能是初次执行且存储中还没有任务的执行结果，所以会调用 computeOrReadCheckpoint 方法（见代码清单 10-28）计算或者从检查点恢复。

小贴士： 这里需要说说 iterator 方法的容错处理过程。如果某个分区任务执行失败，但是其他分区任务执行成功，可以利用 DAGScheduler 对 Stage 重新调度。失败的分区任务将从检查点恢复状态，而那些执行成功的分区任务由于其执行结果已经缓存到存储体系，所以调用 getOrCompute 方法获取即可，不需要再次执行。

代码清单10-27 获取或计算RDD的分区

```
private[spark] def getOrCompute(partition: Partition, context: TaskContext):
    Iterator[T] = {
  val blockId = RDDBlockId(id, partition.index)
  var readCachedBlock = true
  SparkEnv.get.blockManager.getOrElseUpdate(blockId, storageLevel, elementClass-
      Tag, () => {
    readCachedBlock = false
    computeOrReadCheckpoint(partition, context)
  }) match {
    case Left(blockResult) =>
      if (readCachedBlock) {
        val existingMetrics = context.taskMetrics().inputMetrics
        existingMetrics.incBytesRead(blockResult.bytes)
        new InterruptibleIterator[T](context, blockResult.data.asInstanceOf[Iterator-
```

仅供非商业用途或交流学习使用


```
            [T]]) {
          override def next(): T = {
            existingMetrics.incRecordsRead(1)
            delegate.next()
          }
        }
      } else {
        new InterruptibleIterator(context, blockResult.data.asInstanceOf[Iterator[T]])
      }
    case Right(iter) =>
      new InterruptibleIterator(context, iter.asInstanceOf[Iterator[T]])
  }
}
```

根据代码清单 10-27，getOrCompute 方法的执行步骤如下。

1）调用 BlockManager 的 getOrElseUpdate 方法（见代码清单 6-81）先尝试从存储体系中获取 RDD 分区的 Block，否则调用 computeOrReadCheckpoint 方法（见代码清单 10-28）从检查点读取或计算。

2）对 getOrElseUpdate 方法返回的结果进行匹配，将返回的 BlockResult 的 data 属性或返回的 Iterator 封装为 InterruptibleIterator。

computeOrReadCheckpoint 方法在存在检查点时直接从检查点读取数据，否则需要调用 compute 继续计算。computeOrReadCheckpoint 方法的实现如代码清单 10-28 所示。



代码清单10-28　计算或从检查点读取数据

```
private[spark] def computeOrReadCheckpoint(split: Partition, context: Task-
    Context): Iterator[T] =
{
  if (isCheckpointedAndMaterialized) {
    firstParent[T].iterator(split, context)
  } else {
    compute(split, context)
  }
}

private[spark] def isCheckpointedAndMaterialized: Boolean =
checkpointData.exists(_.isCheckpointed)
```

根据代码清单 10-28，computeOrReadCheckpoint 方法的执行步骤如下。

1）如果 checkpointData 中保存了 RDDCheckpointData 且其检查点的状态（cpState）是 Checkpointed，那么调用 firstParent 方法（见代码清单 10-29）找到其父 RDD，然后调用父 RDD 的 iterator 方法。由于 firstParent 中调用了 dependencies，且当前 RDD 的父 RDD 实际是 ReliableCheckpointRDD，那么对 ReliableCheckpointRDD 的 iterator 方法的调用最终将转变为对 ReliableCheckpointRDD 的 compute 方法（见代码清单 10-10）的调用，从而从检查点文件读取之前保存的计算结果。

2）如果 checkpointData 中没有保存 RDDCheckpointData 或其检查点的状态（cpState）

仅供非商业用途或交流学习使用

不是 Checkpointed，那么调用 compute 方法进行计算。

代码清单10-29 找到父亲RDD

```
protected[spark] def firstParent[U: ClassTag]: RDD[U] = {
  dependencies.head.rdd.asInstanceOf[RDD[U]]
}
```

每个 RDD 实现的 compute 方法都不相同。在代码清单 10-10 中曾经介绍了 ReliableCheckpointRDD 的 compute 方法。此处再以 MapPartitionsRDD 和 ShuffledRDD 为例，来看看它们各自实现的 compute 方法。

MapPartitionsRDD 实现的 compute 方法如代码清单 10-30 所示。

代码清单10-30 MapPartitionsRDD的compute方法

```
override def compute(split: Partition, context: TaskContext): Iterator[U] =
  f(context, split.index, firstParent[T].iterator(split, context))
```

ShuffledRDD 实现的 compute 方法如代码清单 10-31 所示。

代码清单10-31 ShuffledRDD的compute方法

```
override def compute(split: Partition, context: TaskContext): Iterator[(K, C)] = {
  val dep = dependencies.head.asInstanceOf[ShuffleDependency[K, V, C]]
  SparkEnv.get.shuffleManager.getReader(dep.shuffleHandle, split.index, split.index
      + 1, context)
    .read()
    .asInstanceOf[Iterator[(K, C)]]
}
```



可以看到，ShuffledRDD 的 compute 方法首先调用 SortShuffleManager 的 getReader 方法（见代码清单 8-96）获取 BlockStoreShuffleReader，然后调用 BlockStoreShuffleReader 的 read 方法（见代码清单 8-90）获取 map 任务输出的 Block 并在 reduce 端进行聚合或排序。

10.5 数据集合 Dataset

Dataset 是特定领域对象的强类型集合，可通过功能或关系操作进行并行转换。当 Dataset 的泛型类型是 Row 时，Dataset 还可以作为 DataFrame。DataFrame 的类型定义如下。

```
type DataFrame = Dataset[Row]
```

有了行的数据集合，DataFrame 看起来更像是关系数据库中的表。DataFrame 是专门为了数据科学应用设计，支持从 KB 到 PB 级的数据量。Spark 支持从文本文件、CSV 文件、Oracle 脚本文件、JSON 文件及所有支持 JDBC 的数据源（例如，MySQL 和 Hive）转换为 DataFrame。

Dataset 中提供的 API 非常丰富，本书不可能逐一进行源码分析。因为 10.8 节将要介绍 word count 的例子，因此本节主要挑选 word count 的例子中使用到的 API 进行分析。

1. ofRows

ofRows 是 Dataset 的伴生对象中提供的方法，用于将逻辑执行计划 LogicalPlan ㊀转换为泛型是 Row 的 Dataset（即 DataFrame）。ofRows 的实现如代码清单 10-32 所示。

代码清单10-32 ofRows方法

```
def ofRows(sparkSession: SparkSession, logicalPlan: LogicalPlan): DataFrame = {
  val qe = sparkSession.sessionState.executePlan(logicalPlan)
  qe.assertAnalyzed()
  new Dataset[Row](sparkSession, qe, RowEncoder(qe.analyzed.schema))
}
```

2. 多种多样的 select

Dataset 提供了多个重载的 select 方法，以实现类似于 SQL 中的 SELECT 语句的选择功能。Dataset 虽然提供了多个 select 操作的 API，但这些 API 最终都将转换为调用代码清单 10-33 展示的 select 方法㊁。

代码清单10-33 Dataset的select方法

```
@scala.annotation.varargs
def select(cols: Column*): DataFrame = withPlan {
  Project(cols.map(_.named), logicalPlan)
}
```

3. rdd

rdd 是 Dataset 的属性之一，由于被关键字 lazy 修饰，因此在需要 rdd 的值时才会进行“懒”执行。rdd 的定义如代码清单 10-34 所示。

代码清单10-34 rdd的定义

```
lazy val rdd: RDD[T] = {
  val objectType = exprEnc.deserializer.dataType
  val deserialized = CatalystSerde.deserialize[T](logicalPlan)
  sparkSession.sessionState.executePlan(deserialized).toRdd.mapPartitions { rows =>
    rows.map(_.get(0, objectType).asInstanceOf[T])
  }
}
```

根据代码清单 10-34，rdd 的语句块执行的步骤如下。

1）SessionState 的 executePlan 方法将逻辑执行计划封装为 QueryExecution ㊂。

2）QueryExecution 的 toRdd 方法将促使执行物理执行计划，并返回 RDD[InternalRow]。

㊀ LogicalPlan 是 Spark SQL 中的对象，用于表示逻辑执行计划。读者可以参阅《深入理解 Spark：核心思想与源码分析》一书中对 Spark SQL 的介绍。

㊁ 由于本书只介绍 Spark 核心的原理，不会对 Spark SQL 中的 Project 作过多介绍。想深入了解 Spark SQL 及 Project 的读者可以阅读笔者所著的《深入理解 Spark：核心思想与源码分析》一书中介绍 Spark SQL 的内容或者市面上其他介绍 Spark SQL 的书籍。

㊂ QueryExecution 是 Spark SQL 中的内容，本书不过多进行介绍。

3）调用 RDD 的 mapPartitions 方法将 RDD 封装为 MapPartitionsRDD。



RDD 的很多方法（如 javaRDD 方法）都会使用 rdd，从而让定义 rdd 的语句块执行。

4. javaRDD

javaRDD 方法用于将 Dataset 内部的 RDD 转换为 JavaRDD，这样就可以用 JavaRDD 提供的 Java API 来操作底层的 Scala RDD 了，其实现如下。

```
def toJavaRDD: JavaRDD[T] = rdd.toJavaRDD()
def javaRDD: JavaRDD[T] = toJavaRDD
```

根据上述代码，javaRDD 方法实际调用了 RDD 的 toJavaRDD 方法（见代码清单 10-20）。

10.6　DataFrameReader 详解

DataFrameReader 用于通过外部数据源的格式（如 text、CVS 等）和数据结构加载 Dataset，DataFrameReader 还提供了非常丰富的操作 DataFrame 的 API。有了 DataFrameReader，我们就可以将各种格式的数据源转换为 Dataset 或 DataFrame，进而以面向关系型数据的方式操纵各种格式的数据。

DataFrameReader 只有三个属性，分别如下。

- source：输入数据源的格式。可通过 spark.sql.sources.default 属性配置，默认为 parquet。

小贴士：Apache Parquet 是 Hadoop 生态圈中一种新型列式存储格式，它可以兼容 Hadoop 生态圈中大多数计算框架（Hadoop、Spark 等），被多种查询引擎支持（Hive、Impala、Drill 等），并且是数据处理框架、数据模型和语言无关的。Parquet 最初是由 Twitter 和 Cloudera（由于 Impala 的缘故）合作开发完成并开源，2015 年 5 月从 Apache 的孵化器里毕业成为 Apache 顶级项目。

- userSpecifiedSchema：用户指定的 Schema，实际的类型是 StructType。
- extraOptions：类型为 HashMap[String, String]，用于保存额外的选项。

DataFrameReader 提供了大量的 API，本书限于篇幅不可能全部都进行分析。由于在 10.8 节将要介绍 word count 的例子，因此本节主要挑选 word count 的例子中使用到的 API 进行分析。

1. format

format 方法（见代码清单 10-35）用于设置输入数据源的格式。

代码清单10-35　DataFrameReader的format方法

```
def format(source: String): DataFrameReader = {
  this.source = source
```

```
    this
  }
```

2. schema

schema 方法（见代码清单 10-36）用于设置用户指定的结构类型（StructType）。

代码清单10-36 DataFrameReader的schema方法

```
def schema(schema: StructType): DataFrameReader = {
  this.userSpecifiedSchema = Option(schema)
  this
}
```

3. 多种多样的 option

DataFrameReader 提供了多个重载的 option 方法，用于向 extraOptions 中添加选项。无论哪个 option 方法，都将转换为对最基本的 option 方法（见代码清单 10-37）的调用。

代码清单10-37 DataFrameReader的option方法

```
def option(key: String, value: String): DataFrameReader = {
  this.extraOptions += (key -> value)
  this
}
```

DataFrameReader 还提供了很多将其他基本类型转换为字符串后添加到 options 中的方法，例如，将长整型转换为字符串后放入 extraOptions 的 option 方法。

```
def option(key: String, value: Long): DataFrameReader = option(key, value.toString)
```

DataFrameReader 还提供了将 scala.collection.Map 或 java.util.Map 中的 key 和 value 作为选项添加到 options 中的方法，例如，将 scala.collection.Map 中的键值对放入 extraOptions 的 option 方法。

```
def options(options: scala.collection.Map[String, String]): DataFrameReader = {
  this.extraOptions ++= options
  this
}
```

4. 多种多样的 load

DataFrameReader 提供了多个重载的 load 方法，用于将数据源的数据加载到 DataFrame。这些 load 方法实际都会转换为调用代码清单 10-38 所示的 load 方法来加载数据。

代码清单10-38 DataFrameReader的load方法

```
@scala.annotation.varargs
def load(paths: String*): DataFrame = {
  sparkSession.baseRelationToDataFrame(
    DataSource.apply(
      sparkSession,
      paths = paths,
      userSpecifiedSchema = userSpecifiedSchema,
```

```
        className = source,
        options = extraOptions.toMap).resolveRelation())
    }
```

根据代码清单 10-38，DataFrameReader 的 load 方法首先构造 DataSource 实例，然后调用 DataSource 的 resolveRelation 方法（见代码清单 10-3）解析得到 BaseRelation，最后调用 SparkSession 的 baseRelationToDataFrame 方法（见代码清单 10-42）将 BaseRelation 转换为 DataFrame。

5. 重载的 text

DataFrameReader 提供了两个重载的 text 方法，用于设置输入数据源的格式为 text 并加载数据，如代码清单 10-39 所示。

代码清单10-39　DataFrameReader的text方法

```
def text(path: String): DataFrame = {
  text(Seq(path): _*)
}

@scala.annotation.varargs
def text(paths: String*): DataFrame = format("text").load(paths : _*)
```

除了 text 方法外，DataFrameReader 还提供了多个重载的 jdbc 方法、json 方法、csv 方法、parquet 方法及 orc 方法，这些方法的实现都与 text 类似，即最终落实到对 format 方法和 load 方法的调用并转换为 DataFrame。

6. 重载的 textFile

DataFrameReader 提供了两个重载的 textFile 方法，用于设置输入数据源的格式并加载数据，如代码清单 10-40 所示。

代码清单10-40　DataFrameReader的textFile方法

```
def textFile(path: String): Dataset[String] = {
  textFile(Seq(path): _*)
}

@scala.annotation.varargs
def textFile(paths: String*): Dataset[String] = {
  if (userSpecifiedSchema.nonEmpty) {
    throw new AnalysisException("User specified schema not supported with
        `textFile`")
  }
  text(paths : _*).select("value").as[String](sparkSession.implicits.newString-
      Encoder)
}
```

可以看到，textFile 方法首先调用 text 方法将文本文件转换为 DataFrame，然后调用 DataFrame 的 select 方法（见代码清单 10-33）进行选择（投影）操作。

10.7 SparkSession 详解

Spark 2.0 引入了 SparkSession，为用户提供了一个统一的切入点来使用 Spark 的各项功能。SparkSession 还提供了 DataFrame 和 Dataset 相关的 API 来编写 Spark 应用程序。SparkSession 降低了学习曲线，使得工程师可以更容易地使用 Spark。

SparkSession 的属性如下。

- sparkContext：即 SparkContext。
- sharedState：在多个 SparkSession 之间共享的状态（包括 SparkContext、缓存的数据、监听器及与外部系统交互的字典信息）。
- sessionState：SparkSession 的状态（SessionState）。SessionState 中保存着 SparkSession 指定的状态信息。
- sqlContext：即 SQLContext。SQLContext 是 Spark SQL 的上下文信息。
- conf：类型为 RuntimeConfig，是 Spark 运行时的配置接口类。

根据 SparkSession 的属性，笔者认为 SparkSession 并非是一个新创造的东西，它不过是对 SparkContext、SQLContext 及 DataFrame 等的一层封装。SparkSession 的创建离不开构建器，在 10.7.1 节将对 SparkSession 的构建器进行分析。在 10.7.2 节，本书将对 SparkSession 提供的 API 进行介绍。

SparkSession 的伴生对象提供了很多属性和方法，以便于我们构造 SparkSession。SparkSession 的伴生对象中包含以下属性。

- sqlListener：类型定义为 AtomicReference[SQLListener]，用于持有 SQLListener。sqlListener 主要用于 SQL UI。
- activeThreadSession：类型定义为 InheritableThreadLocal[SparkSession]，用于持有当前线程的激活的 SparkSession。SparkSession 的伴生对象提供了 setActiveSession 方法、getActiveSession 方法、clearActiveSession 方法分别用于设置、获取、清空 activeThreadSession 中持有的 SparkSession。
- defaultSession：类型定义为 AtomicReference[SparkSession]，用于持有默认的 SparkSession。SparkSession 的伴生对象提供了 setDefaultSession 方法、getDefaultSession 方法、clearDefaultSession 方法分别用于设置、获取、清空 defaultSession 中持有的 SparkSession。

10.7.1 SparkSession 的构建器 Builder

Builder 是 SparkSession 实例的构建器，对 SparkSession 实例的构造都依赖于它。Builder 中的 options（类型为 HashMap[String, String]）用于缓存构建 SparkConf 所需的属性配置。userSuppliedContext 属性用于持有用户提供的 SparkContext，可以通过 Builder 的 sparkContext 方法来设置。Builder 中提供了很多向 options 中增加属性配置的方法，此外还

提供了获取或创建 SparkSession 的方法，本节将分别进行介绍。

1. 多种多样的 config

Builder 提供了多个重载的 config 方法，用于向 options 中添加属性配置。根据声明 options 时指定的泛型，我们知道 Builder 应当提供一个向 options 中添加 key 和 value 都是字符串的方法，代码如下。

```
def config(key: String, value: String): Builder = synchronized {
  options += key -> value
  this
}
```

Builder 还提供了很多将其他基本类型转换为字符串后添加到 options 中的方法，例如，将长整型转换为字符串后放入 options 的 config 方法。

```
def config(key: String, value: Long): Builder = synchronized {
  options += key -> value.toString
  this
}
```

此外，Builder 提供的 appName、master 及 enableHiveSupport，都是通过调用 config 方法实现的。例如，appName 的实现如下。

```
def appName(name: String): Builder = config("spark.app.name", name)
```

2. getOrCreate 方法

getOrCreate 方法用于获取或创建 SparkSession，其实现如代码清单 10-41 所示。

代码清单10-41 Builder的getOrCreate方法

```
def getOrCreate(): SparkSession = synchronized {
  var session = activeThreadSession.get()
  if ((session ne null) && !session.sparkContext.isStopped) {
    options.foreach { case (k, v) => session.sessionState.conf.setConfString(k, v) }
    if (options.nonEmpty) {
      logWarning("Using an existing SparkSession; some configuration may not
          take effect.")
    }
    return session
  }

  SparkSession.synchronized {
    session = defaultSession.get()
    if ((session ne null) && !session.sparkContext.isStopped) {
      options.foreach { case (k, v) => session.sessionState.conf.setConfString
          (k, v) }
      if (options.nonEmpty) {
        logWarning("Using an existing SparkSession; some configuration may not
            take effect.")
      }
      return session
    }
```

```
        val sparkContext = userSuppliedContext.getOrElse {
          val randomAppName = java.util.UUID.randomUUID().toString
          val sparkConf = new SparkConf()
          options.foreach { case (k, v) => sparkConf.set(k, v) }
          if (!sparkConf.contains("spark.app.name")) {
            sparkConf.setAppName(randomAppName)
          }
          val sc = SparkContext.getOrCreate(sparkConf)
          options.foreach { case (k, v) => sc.conf.set(k, v) }
          if (!sc.conf.contains("spark.app.name")) {
            sc.conf.setAppName(randomAppName)
          }
          sc
        }
        session = new SparkSession(sparkContext)
        options.foreach { case (k, v) => session.sessionState.conf.setConfString(k, v) }
        defaultSession.set(session)

        sparkContext.addSparkListener(new SparkListener {
          override def onApplicationEnd(applicationEnd: SparkListenerApplication-
              End): Unit = {
            defaultSession.set(null)
            sqlListener.set(null)
          }
        })
      }

      return session
    }
  }
```

根据代码清单 10-41，getOrCreate 方法的执行步骤如下。

1）如果从 ThreadLocal 中能获取到激活的 SparkSession 且此 SparkSession 还未停止，那么将 options 中的键值对设置到 SparkSession 的 sessionState 的 SQLConf 中并返回此 SparkSession。SQLConf 的作用与 SparkConf 类似，这里不再赘述。

2）如果存在默认的 SparkSession 且此 SparkSession 还未停止，那么将 options 中的键值对设置到 SparkSession 的 sessionState 的 SQLConf 中并返回此 SparkSession。

3）如果用户提供了 SparkContext，则使用 userSuppliedContext 中持有的 SparkContext，否则创建 SparkContext。创建 SparkContext 的操作顺序如下。

① 使用 UUID 生成随机的字符串。当 Application 未指定 spark.app.name 属性时，才以此随机字符串作为 Application 的名称。

② 创建 SparkConf，并将 options 中的键值对保存到 SparkConf 中。

③ 调用 SparkContext 伴生对象的 getOrCreate 方法（见代码清单 4-32）获取或创建 SparkContext。使用此方法获取的 SparkContext 可能是从 SparkContext 的 activeContext 属性中获得的，此 SparkContext 的 SparkConf 还没有保存 options 中的键值对和 Application 的名称，所以需要再次设置。

4）创建 SparkSession，并且将 options 中的键值对设置到 SparkSession 的 sessionState 的 SQLConf 中。此 SparkSession 还将被保存到 defaultSession 中。

5）调用 SparkContext 的 addSparkListener 方法（见代码清单 4-28），向 LiveListenerBus 中添加匿名实现的 SparkListener（此 SparkListener 实现类将在 Application 结束后清空 defaultSession 和 sqlListener）。

6）返回创建或获取的 SparkSession。

10.7.2　SparkSession 的 API

SparkSession 提供了很多 API，限于篇幅不能一一进行介绍。由于在 10.8 节将要介绍 word count 的例子，因此本节主要挑选 word count 的例子中使用到的 API 进行分析。

1. builder

SparkSession 的伴生对象中提供了 builder 方法用于创建 Builder，其实现如下。

```
def builder(): Builder = new Builder
```

2. read

SparkSession 的 read 方法用于创建 DataFrameReader，其实现如下。

```
def read: DataFrameReader = new DataFrameReader(self)
```

3. baseRelationToDataFrame

SparkSession 的 baseRelationToDataFrame 方法用于将 BaseRelation 转换为 DataFrame，其实现如代码清单 10-42 所示。

代码清单10-42　SparkSession的baseRelationToDataFrame方法

```
def baseRelationToDataFrame(baseRelation: BaseRelation): DataFrame = {
  Dataset.ofRows(self, LogicalRelation(baseRelation))
}
```

将 BaseRelation 转换为 DataFrame 后，就可以用操作关系数据的方式来开发。

10.8　word count 例子

本章内容将以 Spark 2.1.0 的 examples 项目自带的 JavaWordCount 为例，从 Java 语言出发，介绍广为人知的 word count，并展示 Spark API 的使用。通过对此例子的介绍，还将帮助读者把调度系统、计算引擎、部署模式等内容串联起来，以对 Spark 核心知识的掌握上升到一个更高的层次。

JavaWordCount 的实现如下。

```
public final class JavaWordCount {
private static final Pattern SPACE = Pattern.compile(" ");
```

```
public static void main(String[] args) throws Exception {

  if (args.length < 1) { // 保证必须有参数，此参数代表待读取文件
    System.err.println("Usage: JavaWordCount <file>");
    System.exit(1);
  }

  SparkSession spark = SparkSession
    .builder() // 创建SparkSession的构建器
    .master("local[1]") // 设置部署模式
    .appName("JavaWordCount") // 设置JavaWordCount例子的应用名称
    .getOrCreate(); // 使用构建器构造SparkSession实例
  // 获取DataFrameReader，使用DataFrameReader将文本文件转换为DataFrame
  JavaRDD<String> lines = spark.read().textFile(args[0]).javaRDD();
  // 使用RDD的flatMap 方法对MapPartitionsRDD进行转换
  JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
    @Override
    public Iterator<String> call(String s) { // 转换函数的作用是对每行文本进行单词拆分
      return Arrays.asList(SPACE.split(s)).iterator();
    }
  });
  // 使用RDD的mapToPair方法对MapPartitionsRDD进行转换
  JavaPairRDD<String, Integer> ones = words.mapToPair(
    new PairFunction<String, String, Integer>() {
      @Override
      // 转换函数的作用是生成每个单词和1的对偶
      public Tuple2<String, Integer> call(String s) {
        return new Tuple2<>(s, 1);
      }
    });
  // 使用RDD的reduceByKey方法对MapPartitionsRDD进行转换
  JavaPairRDD<String, Integer> counts = ones.reduceByKey(
    new Function2<Integer, Integer, Integer>() {
      @Override
      // 转换函数的作用是对每个单词的计数值累加
      public Integer call(Integer i1, Integer i2) {
        return i1 + i2;
      }
    });
  // 使用RDD的collect方法对MapPartitionsRDD及其上游转换进行计算
  List<Tuple2<String, Integer>> output = counts.collect();
  for (Tuple2<?,?> tuple : output) {
    System.out.println(tuple._1() + ": " + tuple._2());
  }
  spark.stop(); // 停止SparkSession
}
}
```

根据 JavaWordCount 的实现，笔者将整个 Job 的提交、执行过程分为以下步骤。

10.8.1 Job 准备阶段

在 JavaWordCount 中，首先对 SparkSession 和 SparkContext 进行初始化，然后通过 DataFrameReader 的 textFile 方法生成 DataFrame，最后调用 RDD 的一系列转换 API 对 RDD 进

行转换并构造出 DAG。下面将对这一过程进行详细的分析。

1. SparkSession 与 SparkContext 的初始化

JavaWordCount 的 main 方法中首先调用 SparkSession 的 builder 方法创建 Builder，然后调用 Builder 的 master 和 appName 两个方法给 Builder 的 options 中添加 spark.master 和 spark.app.name 两个选项，最后调用 Builder 的 getOrCreate 方法（见代码清单 10-41）获取或创建 SparkSession 实例。在实例化 SparkSession 的过程中，如果用户没有指定 SparkContext，那么将创建 SparkContext 并对 SparkContext 初始化。

2. DataFrame 的生成

在创建了 SparkSession 实例后，调用 SparkSession 的 read 方法创建 DataFrameReader 实例，然后调用 DataFrameReader 的 textFile 方法读取参数中指定文件的内容。根据我们对 DataFrameReader 的 textFile 方法的分析，我们知道其实际上调用了 text 方法和 select 方法，而 text 方法又依赖于 format 方法（设置待读取文件的格式）和 load 方法（读取文件的内容）。DataFrameReader 的 load 方法会将 BaseRelation 转换为 Dataset[Row]（即 DataFrame）。

3. RDD 的转换与 DAG 的构建

Dataset 刚被实例化的时候，其属性 rdd 的语句块并未执行，所以当 JavaWordCount 调用 DataSet 的 javaRDD 方法时，会使得 rdd 的语句块（见代码清单 10-34）执行。根据我们对 rdd 语句块的分析，将会调用 QueryExecution 的 toRdd 方法。QueryExecution 的 toRdd 方法将使用 Spark SQL 的执行计划，首先构造 FileScanRDD，然后调用 RDD 的 mapPartitionsWithIndex 方法创建 FileScanRDD 的下游 MapPartitionsRDD，最后调用 RDD 的 mapPartitionsWithIndexInternal 方法创建更下游的 MapPartitionsRDD，完成对 RDD 的部分转换和依赖关系的构建，如图 10-9 所示。

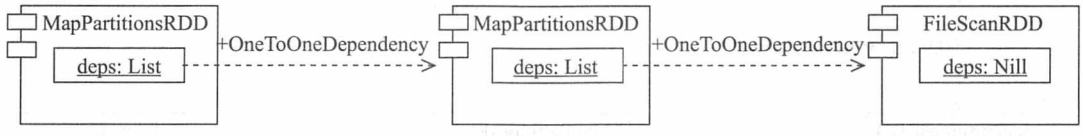


图 10-9 由 Spark SQL 构建出的部分 RDD（即 DAG）



注意 由于 Spark SQL 不属于本书要讲解的内容，所以这里只是简单说明 RDD 的转换与 DAG 构建相关的内容。早期版本的 Spark 中，Spark SQL 与 RDD 的转换及 DAG 的构建是互相分离的部分，现在的版本已经将部分 RDD 转换及 DAG 构建的工作放在了 Spark SQL 中。

在执行完 Spark SQL 的执行计划后，还调用 RDD 的 mapPartitions 方法构造更下游的 MapPartitionsRDD，此时 RDD 的 DAG 如图 10-10 所示。

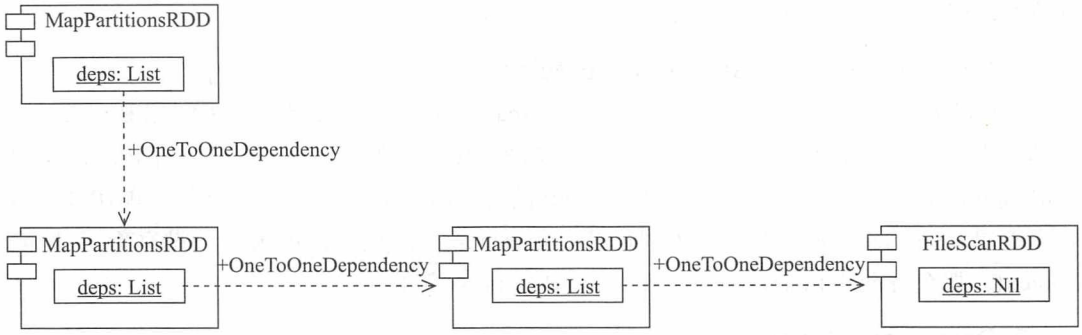


图 10-10　由四个 RDD 构成的 DAG

在调用了 DataSet 的 javaRDD 方法（实际调用 RDD 的 toJavaRDD 方法）后，MapPartitionsRDD 被封装为类型为 JavaRDD 的 lines。

由于 JavaRDD 继承了特质 JavaRDDLike，所以 lines 的 flatMap 方法实际是继承自 JavaRDDLike 的 flatMap 方法。在调用 JavaRDDLike 的 flatMap 方法时，以 FlatMapFunction 的匿名实现类作为函数参数。JavaRDDLike 的 flatMap 方法的实现如下。

```
def flatMap[U](f: FlatMapFunction[T, U]): JavaRDD[U] = {
  def fn: (T) => Iterator[U] = (x: T) => f.call(x).asScala
  JavaRDD.fromRDD(rdd.flatMap(fn)(fakeClassTag[U]))(fakeClassTag[U])
}
```

此时，JavaRDD 内部的 rdd 属性实质是最下游的 MapPartitionsRDD，调用 MapPartitionsRDD 的父类 RDD 的 flatMap 方法（见代码清单 10-18）构造下游的 MapPartitionsRDD，此时 RDD 的 DAG 如图 10-11 所示。

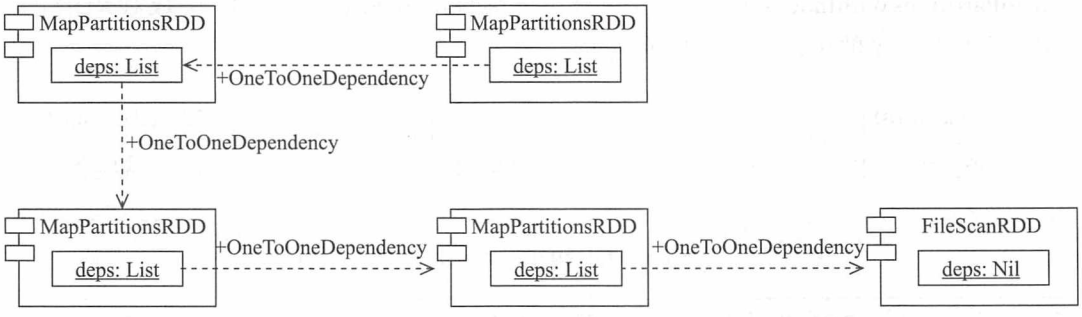


图 10-11　由 5 个 RDD 构成的 DAG

由于变量 words 的类型依然是 JavaRDD，所以调用 words 的 mapToPair 方法其实也继承自特质 JavaRDDLike，其实现如下。

```
def mapToPair[K2, V2](f: PairFunction[T, K2, V2]): JavaPairRDD[K2, V2] = {
  def cm: ClassTag[(K2, V2)] = implicitly[ClassTag[(K2, V2)]]
  new JavaPairRDD(rdd.map[(K2, V2)](f)(cm))(fakeClassTag[K2], fakeClassTag[V2])
```

仅供非商业用途或交流学习使用


```
}
```

根据 mapToPair 的实现，在调用 JavaRDD 内部的 rdd（最下游的 MapPartitionsRDD）的父类 RDD 的 map 方法（见代码清单 10-19）时，以 PairFunction 的匿名实现类作为函数参数，构造下游的 MapPartitionsRDD，并将此 MapPartitionsRDD 封装为 JavaPairRDD，此时 RDD 的 DAG 如图 10-12 所示。

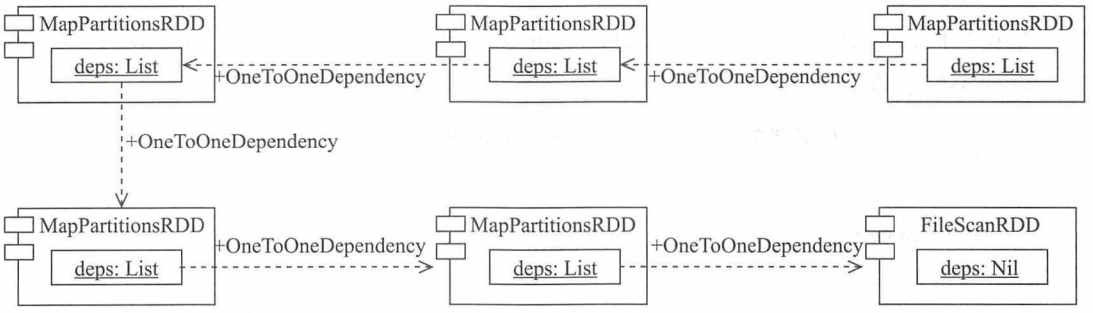


图 10-12 由 6 个 RDD 构成的 DAG

由于变量 ones 的类型为 JavaPairRDD，所以 ones 的 reduceByKey 方法继承自 JavaPairRDD。JavaPairRDD 的 reduceByKey 方法的实现如下。

```
def reduceByKey(func: JFunction2[V, V, V]): JavaPairRDD[K, V] = {
  fromRDD(reduceByKey(defaultPartitioner(rdd), func))
}
```

JavaPairRDD 的 reduceByKey 方法首先调用 defaultPartitioner 方法（见代码清单 10-43）获取默认的分区计算器，然后调用 JavaPairRDD 中重载的另一个 reduceByKey 方法（见代码清单 10-44）。

代码清单10-43 JavaPairRDD的defaultPartitioner方法

```
def defaultPartitioner(rdd: RDD[_], others: RDD[_]*): Partitioner = {
  val rdds = (Seq(rdd) ++ others)
  val hasPartitioner = rdds.filter(_.partitioner.exists(_.numPartitions > 0))
  if (hasPartitioner.nonEmpty) {
    hasPartitioner.maxBy(_.partitions.length).partitioner.get
  } else {
    if (rdd.context.conf.contains("spark.default.parallelism")) {
      new HashPartitioner(rdd.context.defaultParallelism)
    } else {
      new HashPartitioner(rdds.map(_.partitions.length).max)
    }
  }
}
```

根据代码清单 10-43，可以看到 defaultPartitioner 方法的执行逻辑如下。

1）如果 RDD 中有分区计算器，且分区计算器计算得到的分区数量大于零，那么从这

仅供非商业用途或交流学习使用

些分区计算器中挑选分区数量最多的那个分区计算器作为当前 RDD 的分区计算器。

2）如果 RDD 中没有分区计算器，则以 HashPartitioner 作为当前 RDD 的分区计算器。

代码清单10-44 JavaPairRDD的reduceByKey方法

```
def reduceByKey(partitioner: Partitioner, func: JFunction2[V, V, V]): JavaPair-
   RDD[K, V] =
  fromRDD(rdd.reduceByKey(partitioner, func))
```

根据代码清单 10-44，JavaPairRDD 的 reduceByKey 方法将首先调用 PairRDDFunctions 的 reduceByKey 方法（见代码清单 10-45），然后再次封装为 JavaPairRDD。

代码清单10-45 PairRDDFunctions的reduceByKey方法

```
def reduceByKey(partitioner: Partitioner, func: (V, V) => V): RDD[(K, V)] = self.
   withScope {
  combineByKeyWithClassTag[V]((v: V) => v, func, func, partitioner)
}
```

根据代码清单 10-45，PairRDDFunctions 的 reduceByKey 方法将调用 PairRDDFunctions 的 combineByKeyWithClassTag 方法。

PairRDDFunctions 的 combineByKeyWithClassTag 方法的实现如下。



```
def combineByKeyWithClassTag[C](
    createCombiner: V => C,
    mergeValue: (C, V) => C,
    mergeCombiners: (C, C) => C,
    partitioner: Partitioner,
    mapSideCombine: Boolean = true,
    serializer: Serializer = null)(implicit ct: ClassTag[C]): RDD[(K, C)] = self.
        withScope {
  require(mergeCombiners != null, "mergeCombiners must be defined") // required
      as of Spark 0.9.0
  if (keyClass.isArray) {
    if (mapSideCombine) {
      throw new SparkException("Cannot use map-side combining with array keys.")
    }
    if (partitioner.isInstanceOf[HashPartitioner]) {
      throw new SparkException("HashPartitioner cannot partition array keys.")
    }
  }
  val aggregator = new Aggregator[K, V, C](
    self.context.clean(createCombiner),
    self.context.clean(mergeValue),
    self.context.clean(mergeCombiners))
  if (self.partitioner == Some(partitioner)) {
    self.mapPartitions(iter => {
      val context = TaskContext.get()
      new InterruptibleIterator(context, aggregator.combineValuesByKey(iter, con-
          text))
    }, preservesPartitioning = true)
  } else {
    new ShuffledRDD[K, V, C](self, partitioner)
```

仅供非商业用途或交流学习使用


```
        .setSerializer(serializer)
        .setAggregator(aggregator)
        .setMapSideCombine(mapSideCombine)
    }
}
```

根据 combineByKeyWithClassTag 方法的实现，其执行步骤如下。

1）创建聚合器（Aggregator）。

2）如果当前 RDD 的分区计算器与指定的分区计算器相同，则调用 RDD 的 mapPartitions 方法创建 MapPartitionsRDD。

3）如果当前 RDD 的分区计算器与指定的分区计算器不相同，则创建 ShuffledRDD。

在 JavaWordCount 的例子中，调用 combineByKeyWithClassTag 方法将创建 ShuffledRDD。需要注意的是，ShuffledRDD 的 deps 为 null，这是因为 ShuffledRDD 的依赖 ShuffleDependency 是在其 getDependencies 方法（见代码清单 10-46）被调用时才创建的。

代码清单10-46 ShuffleDependency的getDependencies方法

```
override def getDependencies: Seq[Dependency[_]] = {
  val serializer = userSpecifiedSerializer.getOrElse {
    val serializerManager = SparkEnv.get.serializerManager
    if (mapSideCombine) {
      serializerManager.getSerializer(implicitly[ClassTag[K]], implicitly[Class-
          Tag[C]])
    } else {
      serializerManager.getSerializer(implicitly[ClassTag[K]], implicitly[Class-
          Tag[V]])
    }
  }
  List(new ShuffleDependency(prev, part, serializer, keyOrdering, aggregator,
      mapSideCombine))
}
```

此时 RDD 的 DAG 如图 10-13 所示。



由于本例使用了 SparkSesion 的 API 来实现 word count，所以构建了图 10-13 中的 RDD 及 DAG。如果我们采用 SparkContext 的 API 来实现 word count，生成的 RDD 及 DAG 会有所不同（比如会生成 HadoopRDD）。在《深入理解 Spark：核心思想与源码分析》中对使用 SparkContext 的 API 实现的 word count 例子有深入介绍。

10.8.2 Job 的提交与调度

在 JavaWordCount 的最后，调用了动作 API——collect，这将引发对 Job 的提交和调度。Job 的提交与调度大致可以分为 Stage 的划分、ShuffleMapTask 的调度和执行及 ResultTask 的唤起、调度和执行，下面将对这些环节进行深入的分析。

仅供非商业用途或交流学习使用

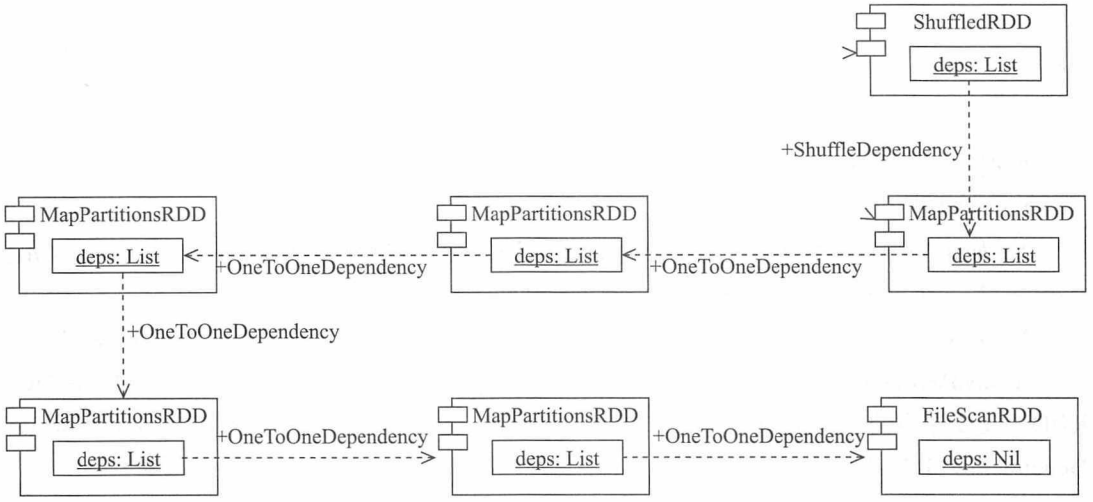


图 10-13　由 7 个 RDD 构成的 DAG

1. Stage 的划分

由于 counts 的类型是 JavaPairRDD，所以调用 counts 的 collect 方法实际继承自父类 AbstractJavaRDDLike，其实现如下。

```
def collect(): JList[T] =
rdd.collect().toSeq.asJava
```

上面的代码中主要调用了 ShuffledRDD 的父类 RDD 的 collect 方法（见代码清单 10-21）。根据 collect 方法的实现，将以图 10-13 所示的由 RDD 组成的 DAG 为参数，调用 SparkContext 的 runJob 方法。SparkContext 的 runJob 方法终将调用代码清单 4-29 中所示的 runJob 方法，进而将 RDD 组成的 DAG 提交给 DAGScheduler 进行调度。根据 7.5 节对 DAGScheduler 的分析，对 DAG 中的 RDD 进行阶段划分后的 Stage 如图 10-14 所示。

图 10-14 中除了 ShuffledRDD 被划入 ResultStage 外，其余的 RDD 都被划入到了 ShuffleMapStage 中。ShuffleMapStage 的 ID 为 0，ResultStage 的 ID 为 1。

2. ShuffleMapTask 的调度与执行

划分完 Stage 后，虽然首先提交 ResultStage，但实际会率先提交 ResultStage 的父 Stage，即 ShuffleMapStage（可回顾 7.4.8 节的分析）。提交 ShuffleMapStage 时会按照分区数目创建多个 ShuffleMapTask，DAGScheduler 将这些 ShuffleMapTask 打包为 TaskSet，通过 TaskSchedulerImpl 的 submitTasks 方法提交给 TaskSchedulerImpl（可回顾 7.4.9 节的分析）。TaskSchedulerImpl 为 TaskSet 创建 TaskSetManager，并将 TaskSetManager 放入调度池，参与到 FIFO 或 Fair 算法中进行调度（可回顾 7.10.4 节的分析）。在被调度后会向 TaskSchedulerImpl 申请资源（可回顾 7.8.2 节、7.10.5 节、9.9.2 节、9.9.5 节的分析），最后将 Task 序列化后封装为 LaunchTask 消息，再发送给 CoarseGrainedExecutorBackend（可

仅供非商业用途或交流学习使用

回顾 9.9.2 节的分析）。CoarseGrainedExecutorBackend 接收到 LaunchTask 消息后将调用 Executor 的 launchTask 方法。Executor 的 launchTask 方法在运行 Task 时将创建 TaskRunner，TaskRunner 实现了 Runnable 接口的 run 方法。TaskRunner 的 run 方法中将调用 Task 的 run 方法，Task 的 run 方法将调用具体 Task 实现类（此时为 ShuffleMapTask）的 runTask 方法（可回顾 9.2.2 节、8.5.2 节的分析）。ShuffleMapTask 经过迭代计算后，将结果通过 SortShuffleWriter 写入磁盘（可回顾 8.5.3 节的分析）。

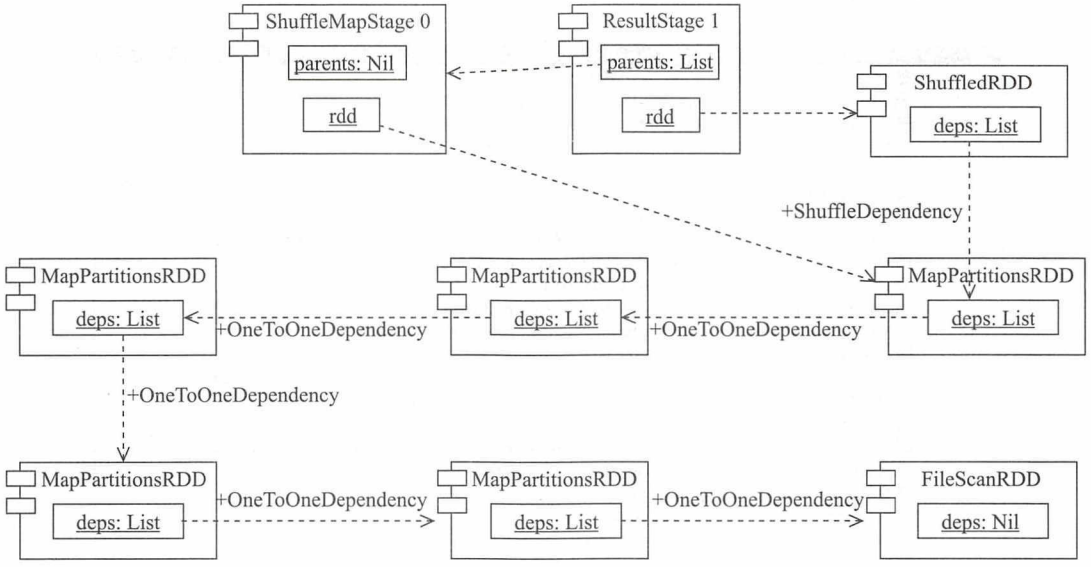


图 10-14 word count 例子最终划分的 Stage

ShuffleMapTask 经过 RDD 管道中对 iterator 和 computeOrReadCheckpoint 的层层调用，最终到达 FileScanRDD（已在 10.4.4 节介绍）。查看此时的线程栈会更直观，如图 10-15 所示。

从图 10-15 看到最底层执行计算的 RDD 是 FileScanRDD，其 compute 方法实际是读取文件列表中每个文件的内容，对其 compute 方法的实现感兴趣的读者可自行查阅。根据对 MapPartitionsRDD 的 compute 方法（见代码清单 10-30）的分析，ShuffleMapTask 将在迭代计算的过程中完成对从文件中读取的每行数据的分词、计数和聚合。

3. ResultTask 的唤起、调度与执行

TaskRunner 将在 ShuffleMapTask 执行成功后调用 SchedulerBackend 的实现类（比如 local 模式下的 LocalSchedulerBackend 或 Standalone 模式下的 StandaloneSchedulerBackend）的 statusUpdate 方法，最终导致 TaskSchedulerImpl 的 statusUpdate 方法被调用。TaskSchedulerImpl 的 statusUpdate 方法（可回顾 7.8.2 节的分析）发现 Task 是执行成功的状态，那么调用 TaskResultGetter 的 enqueueSuccessfulTask 方法获取 ShuffleMapTask 的状态，并将此状

仅供非商业用途或交流学习使用

态交给 DAGScheduler 处理。DAGScheduler 的 taskEnded 方法对于 ShuffleMapTask，需要将 Stage 的 shuffleId 和 outputLocs 中的 MapStatus 注册到 mapOutputTracker。如果有某些分区的 Task 执行失败，则重新提交 ShuffleMapStage，否则调用 submitWaitingChildStages 方法提交当前 ShuffleMapStage 的子 Stage（即 ResultStage）。ResultStage 的提交与调度同 ShuffleMapStage 大致相同，区别有：会按照分区数量创建多个 ResultTask；Task 的 run 方法将调用 ResultTask 的 runTask 方法；ResultTask 经过迭代计算后不会将结果写入磁盘（ResultTask 的迭代计算线程栈如图 10-16 所示）。

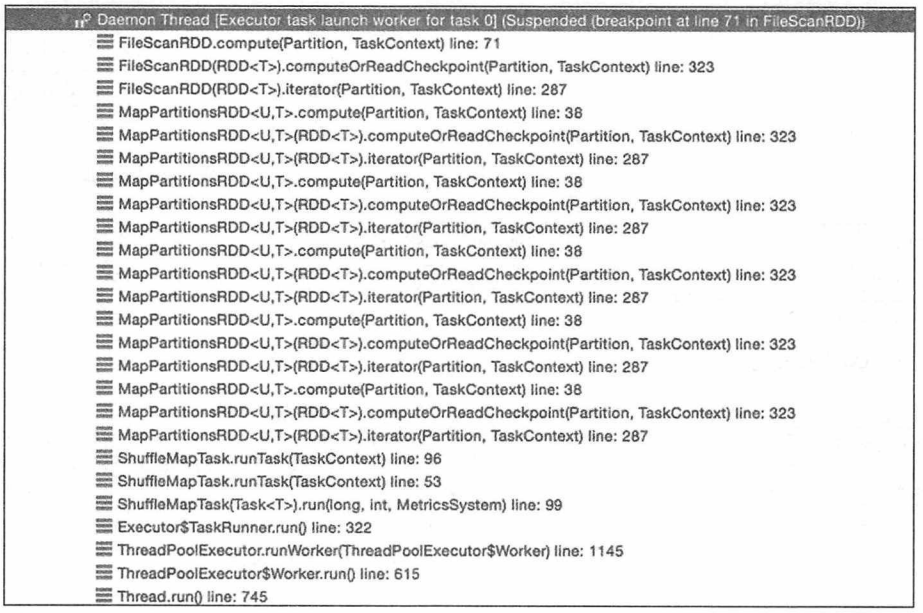


图 10-15　JavaWordCount 例子中 ShuffleMapTask 的迭代计算线程栈

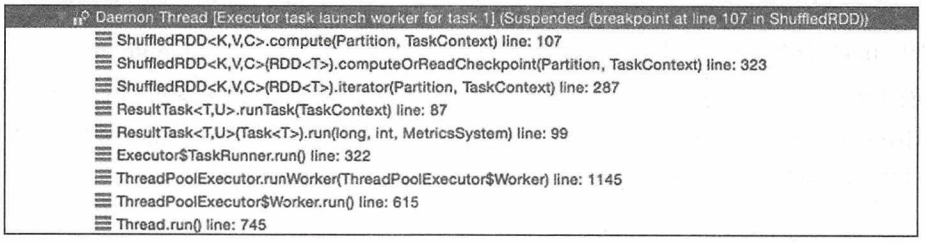


图 10-16　JavaWordCount 例子中 ResultTask 的迭代计算线程栈

根据我们对 ShuffledRDD 的 compute 方法（见代码清单 10-31）的分析，ShuffledRDD 将使用 BlockStoreShuffleReader 的 read 方法（见代码清单 8-90）获取 ShuffleMapTask 输出的 Block 并在 reduce 端进行聚合或排序。ResultTask 执行成功的结果最后也交由 DAGScheduler 的 taskEnded 方法处理，taskEnded 方法中会调用 JobWaiter 的 resultHandler 函数

仅供非商业用途或交流学习使用

将各个 ResultTask 的结果收拢。最后通过 JavaWordCount 例子中的打印语句将整个 Job 的执行结果打印出来，部分打印结果如图 10-17 所示。

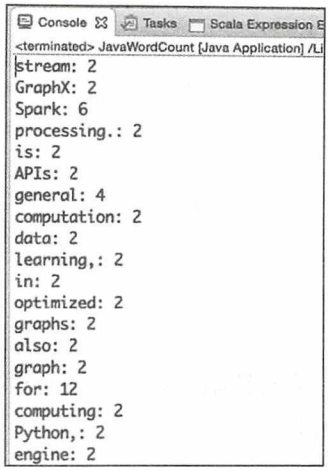


图 10-17 JavaWordCount 例子的输出信息

10.9 小结

经过本章内容的介绍，读者应该理解了什么是 SparkSession。SparkSession 就是通过对 SparkContext、SQLContext 及 DataFrame 的封装，对外提供一致的 API 接口。DataFrame 是 Spark 提供的新 API，专门为数据科学应用设计，以支持更大规模的数据量。DataFrame-Reader 提供了将文本文件、CSV 文件、Oracle 脚本文件、JSON 文件及所有支持 JDBC 的数据源（例如，MySQL 和 Hive）转换为 DataFrame 的 API。

本章最后以 word count 的例子，介绍 Spark API 的基本使用。笔者还通过 word count 的例子，将本书各个章节的内容串联起来，以让大家对应用程序和集群、集群中的各个角色之间的关系有更清楚的理解，并将本书内容有机地结合起来。



附　录

为节省篇幅，附录的所有内容都已经移至笔者本人的博客中。所有附录与博文的对应关系如下。

编　号	标　题	博文链接
A	Spark 核心工具类 Utils	http://blog.csdn.net/beliefer/article/details/77449771
B	Akka 简介	http://blog.csdn.net/beliefer/article/details/77449909
C	Jetty 与 JettyUtils	http://blog.csdn.net/beliefer/article/details/77449938
D	Metrics 简介	http://blog.csdn.net/beliefer/article/details/77450039
E	Hadoop 的 word count 例子	http://blog.csdn.net/beliefer/article/details/77450068
F	命令工具类 CommandUtils	http://blog.csdn.net/beliefer/article/details/77450103
G	Netty 与 NettyUtils	http://blog.csdn.net/beliefer/article/details/77450134
H	RPC 工具类 RpcUtils	http://blog.csdn.net/beliefer/article/details/77450157

作者简介

耿嘉安

10余年IT行业相关经验。先后就职于阿里巴巴、艺龙、360，专注于开源和大数据领域。在大量的工作实践中，对J2EE、JVM、Tomcat、Spring、Hadoop、Spark、MySQL、Redis都有深入研究，尤其喜欢剖析开源项目的源码实现。早期从事J2EE企业级应用开发，对Java相关技术有独到见解。著有《深入理解Spark：核心思想与源码分析》一书。

本书对Spark内部高度抽象的数据结构RDD、分布式DAG调度器/驱动器，以及高效的基于Non-blocking IO分布式计算框架Akka/Netty等内核设计进行了深度剖析，不可多得，是适合大型分布式计算架构师和资深开源贡献者阅读的参考书。

——**蔡栋** 万达网络科技集团总裁助理兼首席数据官、首席架构师

Spark发展的短短几年，以迅雷不及掩耳之势推出RDD、Spark Streaming、Spark SQL、GraphX、MLlib等一系列模块，震撼了大数据圈。这本书结合了最新Spark 2.x版本，在设计思路和代码解析上做了很好的平衡，让开源代码爱好者，喜欢研究源码的同学汲取到一些阅读源码的方法。

——**董飞** datatist首席运营官、前linkedin资深工程师

大数据的书很多，能够写出艺术味道的不多，本书应该可以让你在大数据漫漫征途之中对价值多了一重思考，也可以让你在大数据之巅的惊天骇浪中多了一座灯塔。

——**于俊** 科大讯飞大数据专家

本书为那些想要成为一名合格的Spark工程师，或者致力于成为大数据行业的技术管理人才提供了很好的学习途径。相信读者只要掌握一门Spark技术，就能在大数据的海洋中遨游。感谢笔者为大数据产业做出的贡献！

——**张涵诚** 中关村大数据交易产业联盟副秘书长

本书对Spark原理的讲解与剖析都极具学习意义，作者细致分析了Spark源码的每一个关键细节，对初级用户及中高级用户都有指导意义。

——**王欢** 上海添锡信息技术有限公司技术总监

社区推荐 | CSDN 掘金

投稿热线：(010) 88379604
客服热线：(010) 88379426 88361066
购书热线：(010) 68326294 88379649 68995259

华章网站：www.hzbook.com
网上购书：www.china-pub.com
数字阅读：www.hzmedia.com.cn

上架指导：计算机/大数据分析与处理
ISBN 978-7-111-58439-1
9 787111 584391
定价：139.00元